

The BBC MICRO COMPENDIUM

Jeremy Ruston



Dedicated to Carina Beeson

Tracy

Published in Great Britain by:

INTERFACE, 9-11 Kensington High Street, London W8 5NP.

ISBN 0 907563 33 3

Copyright (C) Jeremy Ruston, 1983.

BASIC ROM contents (C) 1982 Acorn Computer Ltd.

First printing December 1983.

Any enquiries regarding the contents of this book should be directed by mail to the above address.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise, except for the sole use of the purchaser of this book, without the prior written permission of the copyright owner. No warranty in respect of the contents of this book, and their suitability for any purpose, is expressed or implied.

Cover photograph by MRT Studios, London W8.

Typeset and Printed by Commercial Colour Press, London E7.

Contents

Introduction	7
1 Assembly language programming	9
(Diversion: Drawing a mexican hat)	27
(Diversion: Three-dimensional sine curves)	29
2 Computer arithmetic	32
(Diversion: Drawing a milk splash)	41
3 Boolean arithmetic	43
(Diversion: Faster cursor keys)	49
4 Floating point arithmetic	52
(Diversion: Bresenham meets Moire)	66
(Diversion: Blowing up the screen)	68
5 Evaluating expressions	70
(Diversion: Drawing on the tube)	81
(Diversion: Doubling the vertical screen resolution)	83
6 The FROTH threaded language	88
(Diversion: Screen oddity)	114
7 The SLUG structured language	116
(Diversion: Studying DNA)	158
(Diversion: More on Moire)	159
8 Introduction to the BASIC ROM	161
9 The BASIC ROM listing	194

Introduction

This book is a course in advanced programming techniques and algorithms for the BBC Micro.

The first chapter teaches assembly language programming. As well as being useful in its own right, a knowledge of assembly language is vital to understanding the BASIC ROM. The next chapter describes integer arithmetic, which is vital for many assembly language programs. Chapter 3 describes Boolean arithmetic. This is vital in many areas of graphics. Chapter 4 details several algorithms for floating point arithmetic. Using this chapter, the reader has enough material to write a floating point package in assembly language.

Chapter 5 covers expression evaluation. As well as giving a good introduction to the ROM disassembly and SLUG, this section shows some of the practical advantages of recursion. Chapter 6 applies the knowledge of Reverse Polish Notation gained in chapter 5 to write a FORTH like language called FROTH. Unlike a true FORTH, which produces threaded interpreted code, FROTH produces machine code, making it considerably faster than FORTH. Chapter 7 presents a complete compiler for another new language, SLUG. SLUG is an elegant language with some of the features of BCPL, Pascal and Algol. Both SLUG and FROTH are considerably faster than BASIC (from 5 to 100 times faster, depending very much upon the 'benchmark' used). The final two chapters describe and list the BASIC ROM at the heart of the BBC Micro. This serves two purposes. The advanced reader can use it to improve his programming in BASIC, whilst the beginner will find it a useful guide to advanced programming techniques in assembly language.

Thus, the reader of this book will be equipped with the best possible understanding of BBC BASIC and assembly language.

Because of the complex nature of much of the material in this book, I have attempted to lessen the intellectual burden of the reader (and the writer) by providing a number of 'diversions' as interludes between the chapters. A couple of the diversions are trivial, some involve some very complex and attractive graphics, one is a useful utility and one doubles the vertical screen resolution to 512 pixels.

This book was written on a BBC Micro running the Wordwise wordprocessor from Computer Concepts. I am grateful to Kathy Goudge, who typed the original draft, and to John Coll, Chaz Moir, Rob Pickering, Jeremy San, Richard Gollner and Tim Hart-

nell who all provided helpful advice. The proof reading work of Penelope O'Rorke was invaluable. Finally, I must thank David Johnson-Davies, of Acornsoft, who first aroused my interest in languages.

A Note About Basics and Operating Systems

At the time this book was written, there were two versions of BASIC and two versions of the operating system in use on the BBC Micro. This book was written under OS1.2; however all the programs should work on OS1.0 and upwards. Users of OS0.10 should note that some small modifications might be necessary to use some of the programs. However, SLUG and FROTH, the most important programs in the book, will work with any operating system.

I used BASIC II to write the book. BASIC I is very similar, but it has a number of obscure bugs. Naturally, the BASIC II ROM disassembly does not directly apply to BASIC I, but, as I argue in chapter 8, this is not of any importance.

Assembly Language Programming

Many programmers of the BBC micro will restrict themselves to programming in BBC BASIC. This is not a bad idea, since BASIC carries many advantages - programming can generally be done quickly, easily and with less chance of things not working when you try to run the program. Just as importantly, BASIC is an easy language to debug.

Once programmers begin to realise the disadvantages of BASIC - slow speed and long length, they often start using a compiler in order to gain extra speed. Indeed, this book devotes a sizable section to writing compilers. However, the time often comes when you will wish to learn assembly language. This is usually because code generated by a compiler is never as efficient as 'real' assembly language - especially if you choose to compile programs written in BASIC, which is not really a suitable language for compilation. This section is intended to be a complete tutorial course on assembly language. The ROM disassembly and, to a lesser extent, the sections on compilers will improve your programming in a practical way, since they contain assembly language.

A computer is based around a particular microprocessor. In the case of the BBC Micro it is a 6502, but many computers are based on other (usually more exotic) processors, such as the 8088, 8086 or the Z80.

Each of these microprocessors responds to its own particular set of machine level instructions which make it carry out certain very basic operations.

These machine level operations are usually of the order of complexity of "fetch a number from a particular memory location", "add these two numbers together" or "store a number in this memory location".

All these operations are identified by a special number - the operation code, or op-code - which depends on the microprocessor in use. To program a computer in these numbers, one simply fills up memory with the relevant numbers and sets the computer up to execute these numbers.

This is a very tedious way of programming because the numbers themselves bear no relation to the operation being performed. This makes them rather difficult to remember, especially in the case of a machine like the 8086, where there are thousands of different numbers.

Rather than expect programmers to remember these sequences of numbers, manufacturers have developed various mnemonics to represent these numbers.

For example, a common operation in assembly language programming is to load a given number into a variable. (In assembly language there are only three variables available). In machine code, this is represented as &A9 &XX, where &XX is the number to be loaded. On the other hand, the same instruction in 6502 assembly language is LDA #XX.

The 6502 Registers

Internally, the 6502 consists of several registers. These registers are similar to the variables that are used in BASIC except that they are more limited in range. Most 6502 registers are 8 bits wide, which means that they can only hold a number between 0 and 255.

The registers provided are the accumulator, which is often referred to as the A register, the two index registers which are usually referred to as X and Y, the program counter register which is called PC, the stack pointer and the status register.

The accumulator is the most important register of the system. It is used for all arithmetic and logical operations. It is 8 bits wide.

The index registers are normally used to access memory but they are often used for passing parameters to subroutines. The index registers cannot be used for arithmetic or logical instructions. Both index registers are 8 bits wide.

The program counter is 16 bits wide, but it cannot be accessed directly by the programmer. It is used to keep track of which instruction is being executed. It could be looked upon as a book marker.

The strange thing about the stack pointer is that it is only eight bits wide. Because it is used to access memory, we might expect it to be a full 16 bits wide. In fact, it is always assumed to be between addresses &100 and &1FF, so the top eight bits are not stored.

The status register contains 7 bits which reflect various things about the current state of the microprocessor. For example, one of the bits is always a 1 (or 'set') if the last instruction dealt with a number that was zero and is only unset if the number was not zero. These flags are often used for operations like discovering which of two numbers is the larger.

Because the 6502 only has 3 general purpose registers, it is also possible to treat all the memory locations in page zero (the bottom 256 bytes of memory) as 256 extra registers. They aren't quite as good as real registers, since they are slower to access. In addition, in the case of BBC Micro, many of these registers in page zero are used by BASIC and the operating system. In fact, the only locations in page zero that are free for use are &70 to &8F. In practice, neither BASIC I nor BASIC II use memory locations &50 to &6F.

Addressing Modes

Most of the instructions the 6502 responds to access memory in some way in order to get the data needed for the instruction. The way in which they access memory varies according to what the instruction does.

There are 11 basic addressing modes (or ways to access memory). We will look at each one in detail.

The most obvious addressing mode is immediate addressing. In this mode the data for an instruction follows immediately the op-code identifying the instruction. In other words, this mode means that the accumulator should be loaded with a particular number. It is similar to the BASIC statement `LET A=23`. To indicate that this addressing mode is in use, the number is preceded with a #sign (pronounced 'hash').

For example, `LDA #8` would place the number eight in the accumulator.

On the other hand, if you miss out the hash sign, direct memory addressing is used. This means 'load the accumulator with the contents of the given memory location'. Using the above example, `LDA 8` would mean 'load the accumulator with the contents of location 8'. In BASIC, a similar statement would be '`LET A=?8`'. This mode is slower than immediate addressing, since two memory accesses have to be made (one to get the address of the data and one to retrieve the data). Immediate addressing only needs one memory access.

Direct addressing comes in two forms:

8 bit addressing and 16 bit addressing. The idea is that you can access any location using the 16 bit addressing mode (for example, `LDA &3200`). If you wish to access a page zero register, you can omit the top eight bits of the address. This makes the page zero instructions faster than the others. The BBC Micro assembler automatically decides which of the two kinds of direct addressing should be used. Thus, you rarely have to think about the two kinds of instructions. However, they dictate that it is sensible to ensure that any data that you may wish to access frequently during a program should be placed in page zero.

The next addressing mode is sometimes call 'implied' or 'inherent'. In this mode no data is required by the instruction. This means that the instruction is written without any memory address. It doesn't need any data because all the instructions that use this mode imply their own data. Examples of this mode are `CLC`, which clears the carry flag, `TAX`, which transfers the contents of register A to register X, and the similar `TAY`.

Accumulator addressing is similar to implied addressing, except that the data is always assumed to be the accumulator. The trouble is that these instructions can often use other addressing modes, so it becomes necessary to include an indication of the addressing mode required. The normal way of doing this is to include the letter 'A'

after the mnemonic. For example, the instruction `ASL &3200` means 'multiply the contents of location `&3200` by two', whilst `ASL A` means 'multiply the contents of the accumulator by two'.

From now on, the addressing modes become more complex.

The first of these more complex modes is pre-indexed indirect addressing. In all probability, you will never remember the name of this mode, but you will remember how it works.

The format of this mode (using `LDA` as an example) is `LDA (&20,X)`. In this case, the computer first adds together `&20` and the contents of the `X` register. If the answer to this sum is over 256, the computer subtracts 256. It treats this number as an address in page zero. From this address, it retrieves two numbers - one from the address indicated, and one from the next address after the one indicated. The second of these numbers is multiplied by 256 before being added to the first. This new number is treated as the address from where the data for the instruction will be extracted. Quite often, the `X` register will be zero when this mode is used, whereupon, it becomes a simple means for getting the byte pointed to by an address in page zero.

Post-indexed indirect addressing is similar to pre-indexed indirect addressing. In this mode the format is `LDA (&20),Y`. You cannot use this addressing mode with the `X` index register.

Using this mode, a 16 bit number is retrieved from the indicated memory location and the one following it. (In this case, the bottom 8 bits come from the contents of location `&20`, and the top 8 bits come from location `&21`). The contents of the `Y` register is then added to this 16 bit number to gain a new 16 bit number. The data for the instruction is then obtained from the location indicated by this number. This mode is useful in a number of different applications. For example, the indicated memory locations could contain the start of a table. Then it would be easy to access the `Y`th element of the table - assuming the elements of the table were 8 bits wide. Even if they weren't this would still be a trivial program.

Indexed addressing is rather simpler than post-indexed indirect addressing, but the two modes share some common characteristics. Indexed addressing is written as `LDA &20,X`. In this mode, the address of the data for the instruction is `&20 + X` - in other words, the BASIC equivalent of the above would be `LET A = X + &20`. This mode can be used to access tables when you know the address of the table at the time the program is written.

The indirect addressing mode, which can only be used with the `JMP` instruction, is similar to post-indexed indirect addressing. Using this mode, the 16 bit address that the `JMP` instruction must jump to is not given literally, rather an address is given where the actual jump address can be found. For example, the instruction `JMP (&200)`

would pass control to the routine whose address was stored as a 16 bit number in locations &200 and &201. All the operating system routines are accessed using indirect addressing.

Relative addressing is only used with certain types of jump instruction. Using this mode, a constant is added to the program counter, allowing a jump to be made relative to the program counter by a few bytes. The range of relative addressing is limited to about 125 bytes in each direction

The Status Register

The flags in the status register are each 1 bit long. If the bit corresponding to a flag is a '1', that flag is said to be set - otherwise it is unset or reset. Each of these flags reflect various internal states the processor can be in.

The bits in the status register have the following meanings:

- Bit 0 - Carry flag
- Bit 1 - Zero flag
- Bit 2 - Interrupt disable status
- Bit 3 - Decimal mode
- Bit 4 - Break status
- Bit 5 - Not used
- Bit 6 - Overflow flag
- Bit 7 - Sign flag

To examine them in detail:

The carry flag usually consists of the 9th bit of an arithmetic instruction. For example, if 200 and 100 are added together, the result is a number outside the normal range of the accumulator, ie. 300. To get around this problem, the most significant bit of this answer is stored in the carry flag and the rest is stored in the accumulator.

The zero status simply remembers whether the last number dealt with by the processor was zero or not.

The interrupt status flag allows interrupts to be enabled or disabled. If this bit is set, it means that interrupts are disabled and if it is unset it means that interrupts are not disabled - they are enabled. We shall be talking further about interrupts later on, when we look at the instructions which affect this flag.

The decimal mode status is set if decimal mode is in effect and unset otherwise. In decimal mode all arithmetic operations are carried out using decimal arithmetic rather than binary arithmetic. Again, we shall see the significance of this flag when we look at the instructions which affect it and are affected by it.

The break status is not normally used in user programs, since it is only affected by

interrupts, which are handled in the operating system. In brief, the 6502 jumps to the same address when it finds either a break instruction or receives an interrupt. This flag allows the computer to see which of the two caused it to stop what it was doing.

The overflow status reflects the status of bit 6 of the last byte that we have used, while the sign status reflects the value of bit 7. If bit 7 is a 0, the number which is being tested is positive. If it is set, it means the number is negative.

The rest of this section explores the instructions that may be used with the above addressing modes.

The ADC Instruction

This instruction mnemonic means 'add with carry'. The action of the instruction is to add two numbers together, taking into account the current setting of the carry bit. It works in 8 addressing modes:

Immediate

Absolute (to a 16 bit address)

Zero Page (to address in zero page)

Pre-Indexed with Index Register X

Post-Indexed with Index Register Y

Zero Page Indexed with Index Register X

Absolute Indexed with Index Register X

Absolute Indexed with Index Register Y

The ADC instruction retrieves the data from the address indicated, adds it to the accumulator and then finally adds in the contents of the carry flag. As we noted earlier, it copies the state of the imaginary 9th bit of the accumulator to the carry flag.

The important point is that because the carry flag is involved in both ends of the addition, we can add numbers that are larger than the actual size of the accumulator.

But what if we simply want to do a simple addition like $2 + 2$? To demonstrate this, we'll have to introduce an instruction out of the proper order, the LDA instruction. It simply loads a number into the accumulator. So, code to add two and two might be:

```
LDA #2
```

```
ADC #2
```

All this code does is to load the accumulator with 2, then add 2 to the 2 already in the accumulator. No it doesn't. It doesn't because the carry flag is also taken into account. The only way to ensure the carry flag doesn't muck up the sum is to take steps to ensure it is unset before the sum is carried out. This calls for another new instruction, CLC, which clears the carry flag. So all we need to do is add a CLC instruction to the start of the above code.

If you want to add larger numbers, you can do something like this:

1. Clear the carry flag
2. Add together the least significant bytes of the two numbers
3. Add the next bytes in ascending order
4. Repeat step 3 until all the bytes have been added

Using this technique, the carry flag will automatically take care of itself. The net effect is similar to the way some people add multi-digit decimal numbers, writing the carry digit as a small superscript to the original number.

The AND Instruction

This instruction logically ANDs the contents of a memory location with the contents of the accumulator. The AND operation is identical to the AND operation carried out by the BASIC keyword AND. However, the assembly language version of AND only acts on eight bits at a time.

It can easily be extended to act upon data of arbitrary length, in the same sort of way as we extended the ADC instruction, by using more than one AND instruction, each acting upon a different pair of bytes.

The addressing modes allowed with the AND instruction are the same as those used with the ADC instruction.

The AND instruction sets the flags as follows:

Zero flag - set if the result of the calculation was zero.

Sign flag - set if the result was negative (it reflects the status of bit 7 of the result)

The ASL Instruction

The ASL instruction works with rather fewer addressing modes than the ADC and AND instructions.

The addressing modes allowed are:

Accumulator, eg. ASL A

Zero page direct, eg. ASL &20

Absolute direct, eg. ASL &3000

Zero page indexed with X, eg. ASL &20,X

Absolute indexed with X, eg. ASL &3000,X

You'll notice that besides the accumulator mode, these modes can be reduced to two distinct modes - indexed with X and absolute, since the assembler automatically works out whether zero page should be used or not.

The ASL instruction mnemonic stands for 'arithmetic shift left'. This means that the instruction moves all the bits in the number one position to the left. In other words, the

instruction moves the contents of bit 0 to bit 1, bit 1 to 2 and so on. There are some slight problems. Bit zero is going to be undefined and bit 7 has nowhere to go, because bit 8 doesn't exist. In fact, bit zero is always left unset, and the contents of bit 7 are copied into the carry flag, in the same way as the carry flag acts as bit 8 in the ADC instruction.

The other status bits affected are:

Zero flag - set if the result was zero

Sign flag - set if the sign of the result was negative

The BCC Instruction

The BCC instruction is called a conditional jump instruction - or sometimes, a conditional branch instruction. It acts somewhat like the 'IF <condition> THEN GOTO < line__number>' statement of BASIC. The BCC instruction will only carry out the GOTO to a new address if the carry flag is clear.

Rather than loading the program counter with a new value to carry out the branch, it adds a displacement to the present value of the program counter. There are two problems with this approach. The program counter is set to the address after the BCC instruction before the displacement is added to it, and the displacement can only be an eight bit number. This means that the range of the branch is only within +/- 125 bytes of the BCC instruction. Luckily, you don't have to explicitly work out whether a branch instruction will reach a specific address, since the assembler will not assemble an instruction which attempts to branch out of range.

When you are programming normally, all you have to be aware of is the limited range, since the other factors are dealt with automatically.

To use the BCC instruction in your programs, you must follow it with a label. This sample program explains what a label does:

```
.START  
LDA &80  
CMP &81  
BCC START  
RTS
```

This program doesn't do anything useful.

A label is like a place marker in the program. It is created by writing the name of the label preceded by a full stop. (A label can be followed by other instructions without using a colon to start a new statement). When a label is processed by the assembler, it assigns the address of the instruction that follows the label to the variable name given as the label. Thus, labels must adhere to the normal BBC BASIC rules for naming variables. In this way, the label becomes a mnemonic for the address at which it is placed.

When a branch or jump instruction is written, the label following the instruction is taken as the destination for the jump.

It may not seem very useful to be able to execute a jump if the carry flag is set, but it allows us to do several vital things, like see which of two numbers is the larger.

The BCS and BEQ Instructions

The BCS instruction and the BEQ instruction do more or less the same thing as the BCC instruction, except that different conditions spark off the jump. The BCS instruction will only branch if the carry flag is set, whilst the BEQ instruction will only jump if the zero flag is set - in other words, if the last result was zero.

The BIT Instruction

The BIT instruction logically ANDs the contents of the accumulator with the contents of a selected memory location and then sets the condition flags accordingly. Stangely, it doesn't alter the contents of the accumulator or the contents of the memory byte. Thus, the only effect this instruction has is on the condition flags.

The only addressing modes allowed are:

Absolute, eg. BIT &1234

Zero page, eg. BIT &23

In other words, you can only carry out the BIT instruction on the contents of a memory location the address of which is known at the time you write the program.

The point of this instruction is to allow you to see if a certain bit (or bits) of a memory location are set (or unset) without upsetting the contents of the location, and ignoring any untested bits. This is a useful operation since it allows you to set up, in effect, your own flag register in memory which can be used to reflect the status of various things inside your own program.

The way to use it is to select the bits you wish to test of the location. For example, if you wished to see how bit 4 of location &234 was set, the bit in question would be bit 4. Then, turn the 'value' of the bit into a number. The value of bit 4 is 2^4 or 16. You can then write instructions to load this number into the accumulator, then do a BIT instruction with reference to location &234. If the selected bit was zero, the zero flag will be set, otherwise it will be unset. The code needed in this example would be:

```
LDA #16
BIT &234
```

The other use of this instruction is to inspect the contents of bits 6 and 7 of a memory location without disturbing the accumulator. For example, after the above instruction, the sign and overflow flags are set to the state of bits 7 and 6 respectively of location &234. Once these bits have been moved into the flags, you can use them in calculations.

The other result of the action of these two flags is that they allow you to use the top two bits of any location as flags, and then test them without having to do anything to the accumulator - without even having to load a 'mask', as we did above.

To sum up the action of the flags:

Zero flag - set if the result of the AND operation was zero

Sign flag - set to the status of bit 7 of the memory byte selected

Overflow flag - set to the status of bit 6 of the memory byte selected

The BMI, BPL and BNE Instructions

These three branch instructions all act like the BCC instruction, except they branch under different conditions.

The BMI instruction (Branch if MInus) will only branch if the sign bit is set, the BPL instruction (Branch if PPlus) will only branch if the sign bit is unset, and the BNE instruction will only branch if the zero flag is not set.

The BRK Instruction

The BRK instruction is described in the User Guide in its capacity for trapping errors in programs, such as the 'No such line' message in BASIC.

The internal action of the BRK instruction is to set the break flag, push the program counter and status register onto the stack and finally to jump to the routine whose address is contained in locations &FFFE (lsb) and &FFFF (msb). It is worth pointing out that interrupts also jump to the same address. The only way the operating system can see which type of interrupt (BRK or external) caused the jump to the routine is to look at the contents of the flag register. Finally, the action of executing a BRK instruction or responding to an interrupt automatically disables further interrupts.

The BVC and BVS Instructions

The BVC and BVS instructions operate in the same way as the BCC instruction. The BVC instruction, which means Branch if oVerflow Clear, will only carry out a branch instruction if the overflow flag is unset. The BVS instruction means Branch if oVerflow Set and will only carry out a branch instruction if the overflow flag is set.

The CLC Instruction

The CLC instruction simply clears the carry flag and, as we discussed, is often used just before an ADC instruction if the carry flag is not being used in the calculation.

The CLD Instruction

The CLD instruction clears the decimal flag. Under normal conditions, the decimal flag is unset - which implies that binary arithmetic will be carried out.

The CLI Instruction

The CLI instruction clears the interrupt enable flag, in other words enabling interrupts. Under normal circumstances, interrupts are enabled, so this instruction need not be used.

The CLV Instruction

The CLV instruction clears the overflow flag.

The CMP Instruction

The CMP instruction subtracts the contents of a selected memory location from the accumulator and sets the condition flags accordingly, but does not alter the contents of the accumulator or the memory byte. It offers the same memory addressing options as the ADC instruction.

The flags set by the CMP instruction after a sequence of instructions like:

LDA < number__1>

CMP < number__2>

are as follows:

< number__1>	=	< number__2>	Z=1
< number__1>	< >	< number__2>	Z=0
< number__1>	> =	< number__2>	C=1
< number__1>	<	< number__2>	C=0

This table assumes unsigned arithmetic is being used.

Using the above table, you can test numbers to see which is larger, and then use a branch instruction to alter the course of the program depending on some relation.

The difference between signed and unsigned arithmetic is covered in the section on two's complement arithmetic.

The CPX Instruction

The CPX instruction stands for ComPare X register. It is identical in intent to the CMP instruction, with the exception that the X register is used in preference to the accumulator. The addressing modes you can use with this instruction are much more limited:

Immediate, eg. CPX #23

Zero page, eg. CPX &23

Absolute, eg. CPX &2000

This means that you can only compare the contents of the X register to a constant, or to the contents of a memory location whose address is known at the time the program is written/assembled.

The CPY Instruction

The CPY instruction is exactly equivalent to the CPX instruction except that the Y register is used rather than the X register.

The same three addressing modes are used.

The DEC Instruction

The DEC instruction stands for DECrement. It decrements the contents of a location (by 1) and sets various flags accordingly. The addressing modes allowed are:

Zero page direct, eg. DEC &45

Absolute direct, eg. DEC &7C00

Zero page indexed with X, eg. DEC &20,X

Absolute indexed with X, eg. DEC &7C00,X

The flags affected are the sign and zero flags.

The DEX Instruction

The DEX instruction decrements the X register and sets the sign and zero flags according to the result.

The DEY Instruction

The DEY instruction decrements the Y index register and sets the sign and zero flags according to the result.

The EOR Instruction

The EOR instruction Exclusive ORs the contents of the accumulator with the contents of a selected memory location.

It offers the same addressing options as the ADC instruction. The condition flags affected are the sign flag and the zero flag.

The INC Instruction

The INC instruction increments a memory location by 1, but otherwise behaves like the DEC instruction.

The INX and INY Instructions

The INX instruction increments the X register and sets the sign and the zero flags according to the new value in the X register. The INY instruction does the same for the Y register.

The JMP Instruction

THE JMP instruction passes program control to a new address by altering the value in the program counter. It is used with labels in the same way as the branch instructions we looked at earlier.

Two addressing modes are allowed:

Absolute direct, eg. JMP &FFEE

Indirect, eg. JMP (&208)

With indirect jumps, the program counter is loaded with the 16 bit number to be found at the locations indicated (lsb followed by msb).

The JSR Instruction

The JSR instruction is the assembly language equivalent to the BASIC word GOSUB, in that it is used to call subroutines. Similarly, the RTS instruction is the equivalent of RETURN. So, a subroutine in assembly language looks like this:

```
< main program>
JSR < label>
< rest of program>

.< label>
< subroutine code>
RTS
```

The internal action of JSR is quite complex, and need not be understood for normal programming activities. It first pushes the address of the instruction following the JSR instruction onto the stack. This address will be the current contents of the program counter, because the 6502 doesn't process an instruction until the entire instruction has been 'read in'. Finally, it carries out a normal JMP to the address of the subroutine indicated. The RTS instruction simply retrieves the address from the stack, and jumps to it. The idea of using the stack is that it allows you to 'nest' subroutines - have one subroutine being called from inside another. Irritatingly, indirect subroutine calls are not allowed - eg. JSR (&200).

The LDA Instruction

The LDA instruction loads the accumulator from a memory location. The addressing modes allowed are the same as for the ADC instruction. After the accumulator has been loaded, the sign and zero flags are adjusted to reflect the new value in the accumulator.

The LDX Instruction

The LDX instruction loads the index register X from a memory location. The following addressing modes can be used:

Immediate, eg. LDX #&20
 Zero page, eg. LDX &20
 Absolute, eg. LDX &2000
 Zero page indexed with Y, eg. LDX &20,Y
 Absolute indexed with Y, eg. LDX &2000,Y

The sign and zero flags reflect the value loaded into the X register.

The LDY Instruction

The LDY instruction loads the Y register with the contents of a memory location. Again it affects the sign and zero flags. The addressing modes allowed are:

Immediate, eg. LDY #&45
 Zero page, eg. LDY &45
 Absolute, eg. LDY &4500
 Zero page indexed with X, eg. LDY &45,X
 Absolute indexed with X, eg. LDY &4500,X

The LSR Instruction

This instruction moves all the bits in the selected byte one position to the right. It is thus the opposite of ASL. Like the ASL instruction, the previous contents of bit 0 are copied into the carry flag, and zero is copied into bit 7. The sign flag is always unset (think about it), whilst the zero flag is set if the result was zero.

The addressing modes allowed are:

Accumulator, eg. LSR A

Zero page, eg. LSR &56

Absolute, eg. LSR &5678

Zero page indexed with X, eg. LSR &56,X

Absolute indexed with X, eg. LSR &5678,X

The NOP Instruction

The NOP instruction has no effect - which is why it is called 'No Operation'. It is rarely used in assembly programming, but is often useful in machine code programming.

Its only effect is to use up memory, so it can be substituted for instructions you wish to omit from a program in RAM.

The ORA Instruction

The ORA instruction logically ORs the contents of a selected memory location with the accumulator. The addressing modes allowed are the same as for the ADC instruction. Additionally, the condition flags affected are the sign and zero flags.

The PHA Instruction

The PHA instruction, which means PusH Accumulator (onto the stack), does just that. It is often used for restoring return addresses and passing parameters on the stack. No condition flags are affected. Saving the accumulator on the stack is a good way of maintaining its value through a subroutine call. For example, the operating system routines usually retain the value of the accumulator when they pass control back to their calling program. They do this by pushing the accumulator on entry, and pulling it back as soon as they leave.

The PHP Instruction

The PHP instruction pushes the status register on to the stack. Apart from pushing it, this instruction does not affect the status register. It is often used in the same way as the PHA instruction above.

The PLA Instruction

The PLA instruction pulls the accumulator off the stack. Thus, it complements the PHA instruction. It also provides the only means for transferring the contents of the status register to the accumulator - eg.

PHP:

PLA.

The PLP Instruction

The PLP instruction pulls a byte from the stack, then moves it into the status register. The flags are inherently affected.

This instruction is usually paired with a PHP instruction to conserve the status register whilst a subroutine is executing.

The ROL Instruction

This instruction is similar to the ASL instruction. The difference is that when the byte is shifted left, bit zero is not set to zero - rather, it is set to the previous value of the carry flag. The instruction thus rotates the byte, and assumes that the carry flag is the 9th bit of the byte.

The addressing modes allowed are:

Accumulator, eg. ROL A

Zero page, eg. ROL &83

Absolute, eg. ROL &8300

Zero page indexed with X, eg. ROL &83,X

Absolute indexed with X, eg. ROL &8300,X

The idea of this instruction, in part, is to allow the shifting of numbers that are more than eight bits long. For example, if a 32 bit number is stored in &80-&83, the following sequence of instructions will shift the whole lot one position to the left.

ASL &80

ROL &81

ROL &82

ROL &83

In addition to the action of the carry flag mentioned earlier, the sign and zero flags are affected in the normal way.

The ROR Instruction

This instruction is similar to the ROL instruction, except that the rotation is carried out to the right.

Exactly the same addressing options can be used.

The RTI Instruction

The RTI instruction is like the BASIC keyword RETURN, except that it is not used to return from a normal subroutine. Rather, it is used to exit from a subroutine designed to deal with interrupts. Thus, you'll almost certainly never have to use this instruction.

Internally, it pulls the status register off the stack, then pulls the new program counter contents off the stack. Thus, it pulls off the stack exactly what the BRK instruction put there.

You could use the RTI instruction to effect return from a normal subroutine as follows:

```
< main__program>
JSR < label>
< rest of program>

.< label>
PHP
< subroutine code>
RTI
```

In this case, you are substituting RTI for the code PLP, RTS.

The RTS Instruction

This instruction is used like the RETURN keyword of BASIC. It pulls the program counter off the stack, where it was placed by the JSR instruction.

Subroutines are described under the description of the JSR instruction.

The SBC Instruction

This instruction subtracts the contents of the indicated memory location from the accumulator. However, like the ADC instruction, it also takes the contents of the carry flag into account. If the carry flag is set, it is ignored, otherwise, 1 is subtracted from the final answer.

Thus, the SBC instruction is often preceded by an SEC instruction (SEt Carry flag), to ensure that the carry flag does not disturb the result. Like the ADC instruction, bit 8 of the accumulator is assumed to be the carry flag, so if a borrow is necessary (as in 3-5) the carry flag is set.

In keeping with the way the carry flag is treated at the start of the instruction, it is inverted after the instruction. This means that the carry flag will be unset if a borrow was required, and set if it was not.

The SBC instruction can use the same addressing modes as the ADC instruction. Multiple SBC instructions can be concatenated in the same way as multiple ADC instructions.

Besides the carry flag, the sign and zero flags are affected in the normal way by this instruction.

The SEC Instruction

The only effect of this instruction is to set the carry flag.

The SED Instruction

The SED instruction sets the decimal mode flag. This makes the computer carry out decimal arithmetic until the next CLD instruction.

The SEI Instruction

The SEI instruction sets the interrupt disable flag, so disabling interrupts. You can use this instruction in particularly crucial bits of code to ensure that the processor is not interrupted. If you do, you should push the old status value beforehand, and retrieve it afterwards. This ensures that interrupts are treated the same before and after the routine executes.

The STA Instruction

This instruction stores that value of the accumulator to the indicated location. It can use all the ADC addressing modes, except the immediate addressing mode - which wouldn't make any sense in this instruction anyway.

Thus, STA &2000 stores the value in the accumulator to location &2000.

The STX Instruction

This instruction does the same thing as the STA instruction, except it stores the value in the X register. The addressing modes allowed are:

Zero page, eg. STX &80

Absolute, eg. STX &7C00

Zero page indexed with Y, eg. STX &80,Y

Naturally, no flags are affected by this instruction.

The STY Instruction

This instruction stores the value of the Y register to a specified memory location. The addressing modes allowed are:

Zero page, eg. STY &75

Absolute, eg. STY &7500

Zero page indexed with X, eg. STY &75,X

The TAX Instruction

This instruction transfers the value in the accumulator to the X register. In the process, the sign and zero flags are affected in the normal way.

The TAY Instruction

This instruction transfers the value in the accumulator to the Y register. In the process, the sign and zero flags are affected in the normal way.

The TSX Instruction

This instruction transfers the current value of the stack pointer to the X register. It is the only way to examine the contents of the stack pointer.

This instruction is used by the operating system to access information passed on the stack, but normal programming rarely uses it.

The sign and zero flags are affected in the normal way.

The TXA Instruction

This instruction transfers the contents of the X register to the accumulator, affecting the sign and zero flags as it does so.

The TXS Instruction

This instruction copies the X register to the stack pointer, without affecting any flags. It is the normal way to set the stack pointer when the computer is reset. Otherwise, it is rarely used.

The TYA Instruction

This instruction copies the value held in the Y register to the accumulator, affecting the sign and zero flags in the normal way as it does so.

The general form of an assembly language program is:

1. DIM STORE 3000
2. FOR PASS=0 TO 2 STEP 2
3. P%=STORE
4. [OPT PASS
5. < ASSEMBLY LANGUAGE STATEMENTS>
6.]NEXT

(The numbers above are not intended to be line numbers).

We will look at the lines above one by one:

1. This line reserves 3001 bytes of free space. The variable SPACE contains the address of the first byte. The space is used to hold the machine code. 3000 bytes is an arbitrary figure, and could be reduced if the rest of the program gets too big. However, it is your responsibility to ensure that the machine code does not overrun the space allocated to it.
2. This line loops through two values for the OPT statement.
3. The lefthand square bracket is used to enter the assembler. The OPT statement is called a pseudo-operation, since it is a 'fake' assembly language operation. It is described in the User Guide, but the above shows a typical example of its use. On the first pass through the assembly language statements no listing is issued, and errors are suppressed. On the second, errors are not suppressed.
5. This line represents the assembly language statements of the program.

6. This line terminates the program by leaving the assembler and terminating the PASS loop. This ensures that the assembly language is assembled twice.

Some practical examples of assembly language programming are to be found elsewhere in the book, notably in the ROM disassembly.

“Diversión”, “Drawing a mexican hat”

This simple program draws a three dimensional view of a function. The accompanying screen dumps show the output of the program.

The most complex part of the algorithm is the procedure PROCP(K%,X%,Y%,Z%). This acts in the same way as PLOT K%,X%,Y%, except it plots in three dimensions. The routine rotates the point in the horizontal plane before it is plotted. This is achieved using:

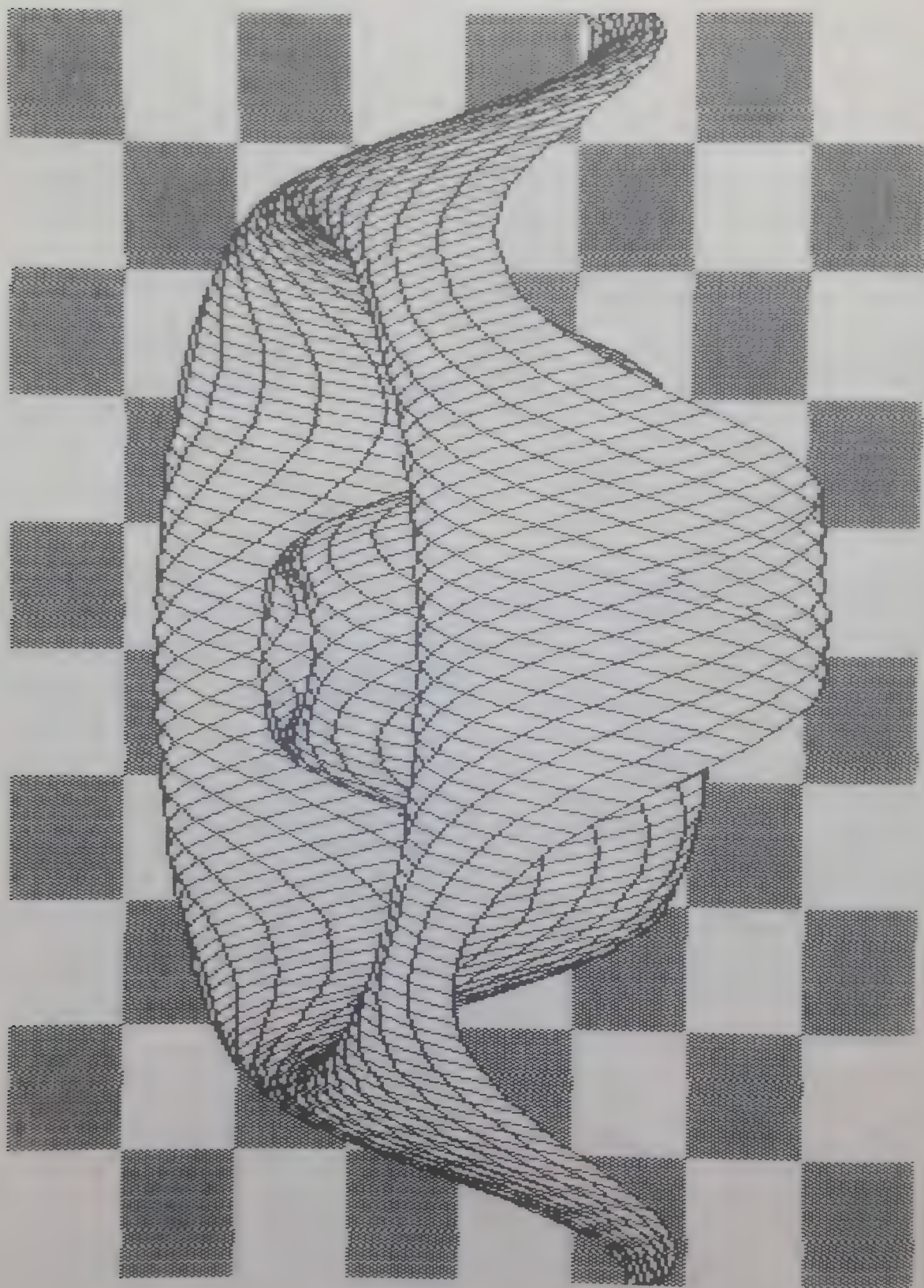
$$X = X * \cos(\text{angle}) + Y * \sin(\text{angle})$$

$$Y = -X * \sin(\text{angle}) + Y * \cos(\text{angle})$$

SIN(angle) and COS(angle) are computed in advance, since the angle of rotation is constant. To plot the rotated point, the Y coordinate is scaled down and summed with the Z coordinate. This makes the plot slightly angled away from the viewer, rising above the horizon the farther away it gets. The rotation factor generally enhances the effect. It should be noticed that this is not a truly general purpose plotting routine, since the squares nearest the viewer appear the same size as those farther away. We shall examine a routine later that passes this criterion.

You may like to alter the function that is plotted. This can be done by altering line 410. This line must yield a result between about -500 and 500 as D% goes from 0 to about 700.

```
> LIST
10 REM Filename:HAT
20
30 REM Mexican hat drawing program
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 VDU 23,224,&AA,&55,&AA,&55,&AA,&55,&AA,&55
90 FOR Y% = 0 TO 31
100 FOR X% = 0 TO 79
110 IF (X% AND 8) <> (Y% AND 4)*2 THEN VDU 31,X%,Y%,224
120 NEXT ,
```

```

130 VDU 30
140 ANGLE% = 40
150 S% = 20
160 C = COS(RAD(ANGLE%))
170 S = SIN(RAD(ANGLE%))
180 VDU 29,640;512;
190 FOR X% = -500 TO 500 STEP S%
200 FOR Y% = 500 TO -500 STEP -S%
210 PROCP(4,X%,Y%,FNA(X%,Y%))
220 PROCP(4,X%,Y% + S%,FNA(X%,Y% + S%))
230 PROCP(87,X% + S%,Y%,FNA(X% + S%,Y%))
240 PROCP(87,X% + S%,Y% + S%,FNA(X% + S%,Y% + S%))
250 PROCP(4,X%,Y%,FNA(X%,Y%))
260 PROCP(5,X% + S%,Y%,FNA(X% + S%,Y%))
270 PROCP(5,X% + S%,Y% + S%,FNA(X% + S%,Y% + S%))
280 PROCP(5,X%,Y% + S%,FNA(X%,Y% + S%))
290 PROCP(5,X%,Y%,FNA(X%,Y%))
300 NEXT Y%
310 NEXT X%
320 *SAVE P.HAT 3000 8000
330 END
340
350 DEF PROCP(K%,X%,Y%,Z%)
360 PLOT K%,X%*C + Y%*S,(-X%*S + Y%*C) DIV 3 + Z%
370 ENDPROC
380
390 DEF FNA(X%,Y%)
400 D% = SQR(X%*X% + Y%*Y%)
410 = SIN(RAD(D%))*200

```

>

“Diversion”, “Three dimensional sine curves”.

This program draws a simple pattern comprised of a family of overlapping sine curves. The screen dump shows the output from it.

The program is valuable since it shows how a three dimensional effect can be built up by overwriting parts of the pattern with new lines.

By changing the equation in line 110 to something more complex, this program could generate quite intricate patterns. The only disadvantage is that it is rather slow.

> LIST

```

10 REM Filename:SINE
20

```




```
30 REM Overlaid sine waves
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 FOR Y% = 400 TO 0 STEP -8
90 GCOL 0,(Y% DIV 8) MOD 2
100 FOR X% = 0 TO 1279 STEP 2
110 MOVE X%,SIN(RAD(Y% + X%))*Y% + Y%*2
120 PLOT 1,0,-100
130 NEXT ,
```

>

Computer Arithmetic

As you know, because a byte contains 8 bits it can only represent one of 256 possible states or, more normally, a number between 0 and 255 inclusive.

Similarly, if we combine 2 bytes to form a 16 bit number we can only represent numbers between 0 and 65535 inclusive. This range is very restrictive because it does **not allow for negative numbers**.

A scheme has been developed to allow the representation of negative numbers using the binary system. The way in which it works is to set aside the most significant bit of the number to indicate whether the number is positive or negative. The normal convention is to assume that if the most significant bit is 0, then the number is positive. If it is 1 then it is negative.

As well as setting the most significant bit to be 1, one other change has to be made to a negative number. All the bits of the number have to be inverted and then 1 has to be added to the number.

For example, assume we wish to encode the number -10 into a single byte using this method. First we encode 10 into a binary number, getting 00001010. To convert it from being a positive number to negative, we have to invert it. This gives us 11110101. We must then add 1 to it, whereupon we get 11110110.

Notice that we don't have to make sure that the top bit is set to one explicitly, since the inversion operation takes care of this automatically.

This system for representing numbers is called the 2's complement system. It is worth bearing in mind that the range of numbers you can represent using 2's complement numbers in a byte is different from the normal range of 0 to 255. The largest number is obviously 01111111 (127 in decimal), since this is the largest number without the top bit set. The smallest number we can represent is 10000000 since this is the smallest number with the top bit set. As an exercise, this is how to convert it to a normal decimal number. First, we invert it, getting 01111111. Then one is added to it, getting 10000000 again. Thus, this number corresponds to -128.

The reason why this system is so powerful is that it allows us to add and subtract numbers regardless of the sign. This can be verified by examining the sample sum shown below:

Add -3 and 23:

Convert -3 to binary:

$$3 = 0000\ 0011$$

$$\text{NOT } 3 = 1111\ 1100$$

$$-3 = 1111\ 1101$$

Convert 23 to binary:

$$23 = 0001\ 0111$$

Add:

$$\begin{array}{r} 1111\ 1101 \\ + 0001\ 0111 \\ \hline 0001\ 0100 \quad (20 \text{ in decimal}) \\ 1\ 1111\ 111 \quad (\text{this row shows the carries}) \end{array}$$

So, the answer comes out as 20, which is correct.

The 2's complement system cannot directly cope with multiplication or division. If you wish to multiply two 2's complement numbers together you must first work out the signs of the two numbers that you are multiplying and then take the absolute values of both numbers. If both of the signs are the same, the answer will be positive. If they are different, the answer will be negative. A similar rule holds for division. If the answer to a multiplication or division is found to be negative, then you must negate the answer either by inverting it and adding 1 to it, or by subtracting it from 0.

Most of the 2's complement arithmetic contained in this book is transparent to the reader but there are some sections such as those in compiler for the FROTH language towards the end of the book which assume a certain amount of knowledge of 2's complement numbers.

One stumbling block in learning assembly language is the absence of some of the features that one takes for granted in BASIC. These include instructions for such simple operations as multiplication and division and program control statements such as FOR - NEXT and REPEAT - UNTIL loops. This section goes some of the way to removing the problem by discussing how to multiply in assembly language.

There are four 6502 instructions which shift bytes left or right:

ASL shifts the contents of a memory location or the accumulator one bit to the left. The least significant bit assumes a value of zero. For example, the binary number 01101011 (6B hex, 107 decimal) would turn into 11010110 (D6 hex, 214 decimal) after an ASL instruction had been performed on it. The most significant bit (the first digit written down) of the original number is copied into the carry flag. The clever thing about the ASL instruction is that the process of shifting a number left is exactly the same multiplying the number by two. So the ASL instruction is the seminal multiplication instruction.

LSR is the opposite of the ASL instruction, in that it shifts a byte one position to the right. The most significant bit becomes zero and the least significant bit is copied into the carry flag. In consequence of this, it divides a byte by two, leaving the remainder in the carry flag.

ROL is identical to ASL, except that the least significant bit is not automatically set to zero, but instead the current status of the carry flag is used.

ROR is identical to LSR, except that the most significant bit is set to the carry flag.

The useful property of these last two instructions is that they allow you to shift numbers which are bigger than a single byte wide. For example, to shift left the four bytes starting at address &80, you could use:

ASL &80

ROL &81

ROL &82

ROL &83

The ASL instruction as it stands can be used to multiply by 2, 4, 8, 16 and so on. This can be done because if you follow one ASL instruction by another, the effect of each is compounded. For example, three ASLs will multiply by 2 three times, giving $2 \times 2 \times 2$. This is a total multiplication factor of eight. If you continued along these lines you would get factors of 16 and 32, and so on. This is great, but many times you may wish to multiply by a number other than 2, 4, 8, 16 and 32 etc. This is relatively easy to do. Let us assume we wish the computer to work out $5 \times X$. You might be able to see that this sum can also be written as $4 \times X + X$. This may not look like much of a simplification, but it is, because in this new form the only multiplication we need to carry out is $X \times 4$, which we know how to do. Therefore to multiply a number by 5, one need only multiply by 4 using the arithmetic shift left instruction twice and then add on to that the original number.

This technique can be extended to apply to other multiplication factors. For example, we can represent a multiplication by 7 as $4 \times X + 2 \times X + X$. Again, this sum can quite easily be coded in assembly language.

We may not appear to be any closer to a fully general multiply routine which will multiply any two numbers together but we have come somewhere nearer that point.

Part of the solution is to examine how we broke down a complicated multiplication, which we couldn't do, into a simpler calculation that we can do.

We started off with the sum $5 \times X$. Notice two things: five in binary is 0101; and the weightings assigned to each binary digit position start at 1 and ascend through 2, 4, 8 etc, multiplying by two each time we move one position to the left.

The interesting thing to note is that when we were doing multiplication in terms of

shifts and additions, we multiplied by 4 and then by 1 for a multiplication by five. There is a connection between what you must multiply X by and the binary representation of the number that is being multiplied. We can now say that to multiply any two arbitrarily long binary numbers we must scan one of the numbers and for every position in which there is a 1 in the number (a bit is set), we must multiply the other number in the calculation by the weighting assigned to the original bit position. We must then add together the results of these calculations. In other words, if the detected bit has a 'value' of 16, you must multiply the other number by 16 and add this result to the grand total.

You can build a perfectly reasonable multiplication routine by using this technique. However, it is usually easier to use the fact that a given multiplication will also do an arithmetic shift left in the process of doing multiplication, which is a complicated way of explaining that if you have two eight bit numbers X and Y the multiplication process should go something like this:

1. The result will be stored in location Z.
2. Do an arithmetic shift left (use the ASL instruction, combined with ROL instructions if the numbers are more than a byte long) of the number X. If the bit that falls off the end of X, in other words the bit that ends up in the carry, is set then set Z to Z + Y.
3. Shift left Z.
4. Repeat the process until X becomes zero.

The program on the next page shows how to do this in assembly language. It will multiply two eight bit numbers together. (Unsigned arithmetic is used, but that's another story).

The program as presented here will not detect overflow (as in 200×200), but this is relatively easy to add. All you need to do to add overflow detection is check to see if $Z + Y$ is more than 255 - this can be checked by examining the carry bit after the addition. It is interesting to note that if you do decide to detect overflow, you can assume that one of the two numbers will be less than 16, since if they were both 16 or greater, overflow would be bound to result. Thus, it would be possible to alter the range of the loop in the program to 4, but you would have to swap around the numbers N1 and N2, according to which had the top four bits set to zero, and take appropriate action if neither did.

An entirely different way of illustrating the process of binary multiplication goes like this:

Call your two numbers X and Y, and assume the answer will appear in a number called N.

First write X in binary, omitting any leading zeros. (In other words, write 101101

rather than 00101101). Then replace each '1' with the letters MA and each '0' with the letter M. A string of Ms and As will then be formed. For example, the binary number above will yield the string:

MAAMAMAAMA

(Any budding blues singers, start singing).

Then set N to zero and scan the string from left to right. For each letter M encountered, multiply N by two. For each letter A, add Y to N, storing the result in N.

When no more letters are left, the number N is the answer. (The same method can be altered to show the raising to a power process in binary. This is described in a subsequent section).

Some analysis will show that this is exactly the same method as the one described earlier.

A decimal form of this method was used by some early civilisations as an easy way to multiply, in preference to long multiplication.

(The brackets in line 180 are necessary to prevent an obscure bug/feature in BBC BASIC where if you carry out ASL (or ROL, ROR etc) on a label starting with 'A', the system interprets you to be using accumulator addressing. This means that 'ASL ANS' would be treated as 'ASL A/NS' !! You can see how this occurs in the ROM listing)

> LIST

```

10 REM Filename:8MULT
20
30 REM 8 bit binary multiplication
40
50 REM (c) 1983 Jeremy Ruston
60
70 DIM SPACE 100
80 N1 = &80:REM Storage for first number
90 N2 = &81:REM Storage for second number
100 ANS = &82:REM Storage for answer
110 FOR LOOP = 0 TO 2 STEP 2
120 P% = SPACE
130 [OPT LOOP
140
150.MULTIPLY
160 LDA #0           /Zero answer
170 STA ANS
180 LDX #8           /Set up loop
190.START

```

```

200 ASL (ANS)           /Multiply answer
210 ASL N1              /Copy bit 7 to carry
220 BCC NO__ADD         /If the bit was zero, skip addition
230 LDA N2              /Get addition factor
240 CLC                 /Clear carry for addition
250 ADC ANS             /Add into answer
260 STA ANS             /Store back to answer
270.NO__ADD
280 DEX                 /Check for end of loop
290 BNE START
300 RTS
310 ] NEXT LOOP
320
330 REPEAT
340 INPUT "Enter two numbers:"?N1,?N2
350 CALL MULTIPLY
360 PRINT "Answer is ";?ANS
370 UNTIL FALSE

```

>

The program starts by reserving space for the assembly language at the label SPACE.

Lines 80 to 100 simply assign labels to the zero page locations used by the program.

Lines 110 to 130 are the mechanism for making two passes over the assembly language.

The program starts at the label MULTIPLY. First, it sets the answer location to zero. This has to be done because the answer is accumulative - that is, it is arrived at through a sequence of additions to the answer location. Next the X register is initialised to 8. This is because we shall be making 8 iterations of the main loop, to allow for eight bits in the two original numbers.

The first 'active' piece of code occurs at line 200, where the answer is multiplied by two.

Next, bit seven of one of the two original numbers is copied into the carry flag. This is done by means of an ASL instruction. A cause of confusion is that this ASL instruction is not being used as a device to multiply by two with.

Next, if no carry was generated, the code for adding the other number and the answer are skipped.

Finally, the loop is terminated in the normal way.

As a matter of interest, here is a version of the above program written using all the improvements I could think of. It should help to demonstrate tactics you can use in cleaning up your own programs.

> LIST

```

10 REM Filename:8MULT2
20
30 REM Better 8 bit binary multiplication
40
50 REM (c) 1983 Jeremy Ruston
60
70 DIM SPACE 100
80 N1 = &80:REM Storage for first number
90 N2 = &81:REM Storage for second number
100 ANS = &82:REM Storage for answer
110 FOR LOOP = 0 TO 2 STEP 2
120 P% = SPACE
130 [ OPT LOOP
140
150.MULTIPLY
160 LDX #8           /Set up loop
170 LDA #0           /Zero answer
180.START
190 ASL A             /Multiply answer
200 ASL N1             /Copy bit 7 to carry
210 BCC NO__ADD       /If the bit was zero, skip addition
220 CLC               /Clear carry for addition
230 ADC N2             /Add into answer
240.NO__ADD
250 DEX               /Check for end of loop
260 BNE START
270 STA ANS           /Store answer
280 RTS
290 ] NEXT LOOP
300
310 REPEAT
320 INPUT "Enter two numbers: "?N1,?N2
330 CALL MULTIPLY
340 PRINT "Answer is ";?ANS
350 UNTIL FALSE

```

>

As you can see, the main speed improvement has come from replacing some memory accesses by register accesses.

It is also fairly easy to take the powers of numbers in assembly language. (For example, 2 'to the power of' 3 is two multiplied by itself three times which comes to eight. In BASIC, this is 'PRINT 2^3').

There are several reasons why we should wish to be able to compute powers of numbers. A simple compiler or interpreter that only supports integer arithmetic can use this method to implement powers, without needing floating point arithmetic. I find the most common use of the power operator is to help in extracting bits from a byte. In addition, many arithmetic and numerical analysis techniques require power calculations. These techniques can be speeded considerably by using a fast algorithm such as this.

For example, we shall examine how to compute X to the power of N , given X and N , and assuming N is a positive integer.

Let us assume that N is 16. We could start with X and multiply it by itself 15 times. This is the obvious way to do it, but it is needlessly complex and slow. It is possible to obtain the same answer with only 4 multiplications, as opposed to fifteen. The method is to repeatedly take the square of each partial result. This will yield the partial answers X^2 , X^4 , X^8 and X^{16} . This result is extracted from one of the basic laws of indices, which state that $(X^N)^M$ is the same as $X^{(N \cdot M)}$.

The same idea can be applied to any value of N in the following way:

1. Write the number N down in binary, but omit any zeros on the left, ie. the first digit must be a 1.
2. Replace each 1 in the number by the pair of letters SX and replace each zero by the letter S .
3. Cross off any SX pairs that appear on the left.
4. The result is a sequence of the letters S and X . Oddly enough, this result can be used for computing X to the power of N .
5. S is interpreted as the operation of squaring and X is interpreted as the operation of multiplying by X .

As an example, I shall work through the above method if N is equal to 23.

The binary representation of 23 is 10111. This gives a letter sequence of $SX S SX SX SX$. We can remove the leading SX to gain the answer $S SX SX SX$.

This rule states that we should square the number twice, then multiply by X , square it again, multiply by X , square it and then multiply by X .

We would be successively computing X^2 , X^4 , X^5 , X^{10} , X^{11} , X^{22} and X^{23} . This binary method is pretty easy to translate into assembly language as long as you have a suitable multiplication routine, like those we have discussed previously.

A computer program to do all this often bears very little resemblance to the above algorithm. The method used is as follows:

To find X^N :

1. Set Y to 1 and Z to X.
2. Shift N right. If the bit that fell off was zero, go on to step 5.
3. Set $Y = Z * Y$.
4. If $N = 0$, the program has finished; the answer is Y.
5. Set $Z = Z * Z$.
6. Go back to step 2.

This can be encoded in the following simple BASIC program:

```

10 REM POWER
20
30 REM Binary method for exponentiation
40
50 REM (c) 1983 Jeremy Ruston
60
70 INPUT "What do you want to the power of what:"Z,N
80
90 REM Step 1:
100 Y = 1
110
120 REM Step 2:
130 N = N/2
140 IF N = INT(N) THEN GOTO 230
150 N = INT(N)
160
170 REM Step 3:
180 Y = Z*Y
190
200 REM Step 4:
210 IF N = 0 THEN PRINT "Answer:"Y:END
220
230 REM Step 5:
240 Z = Z*Z
250
260 REM Step 6:
270 GOTO 120

```

You can trace through this program by hand to see exactly how the algorithm works. You may also like to encode the program into assembly language. If you decide to do so, I would recommend you stick a limit of one byte on the lengths of all the variables used.

“Diversion”, “Drawing a milk splash”

This program draws a pattern on the screen resembling a milk drop splashing into a cup. The screen dump shows the effect.

The program is extremely slow to run, so it makes sense to *SAVE the result as the last line of the program, so that it can be reproduced quicker in future.

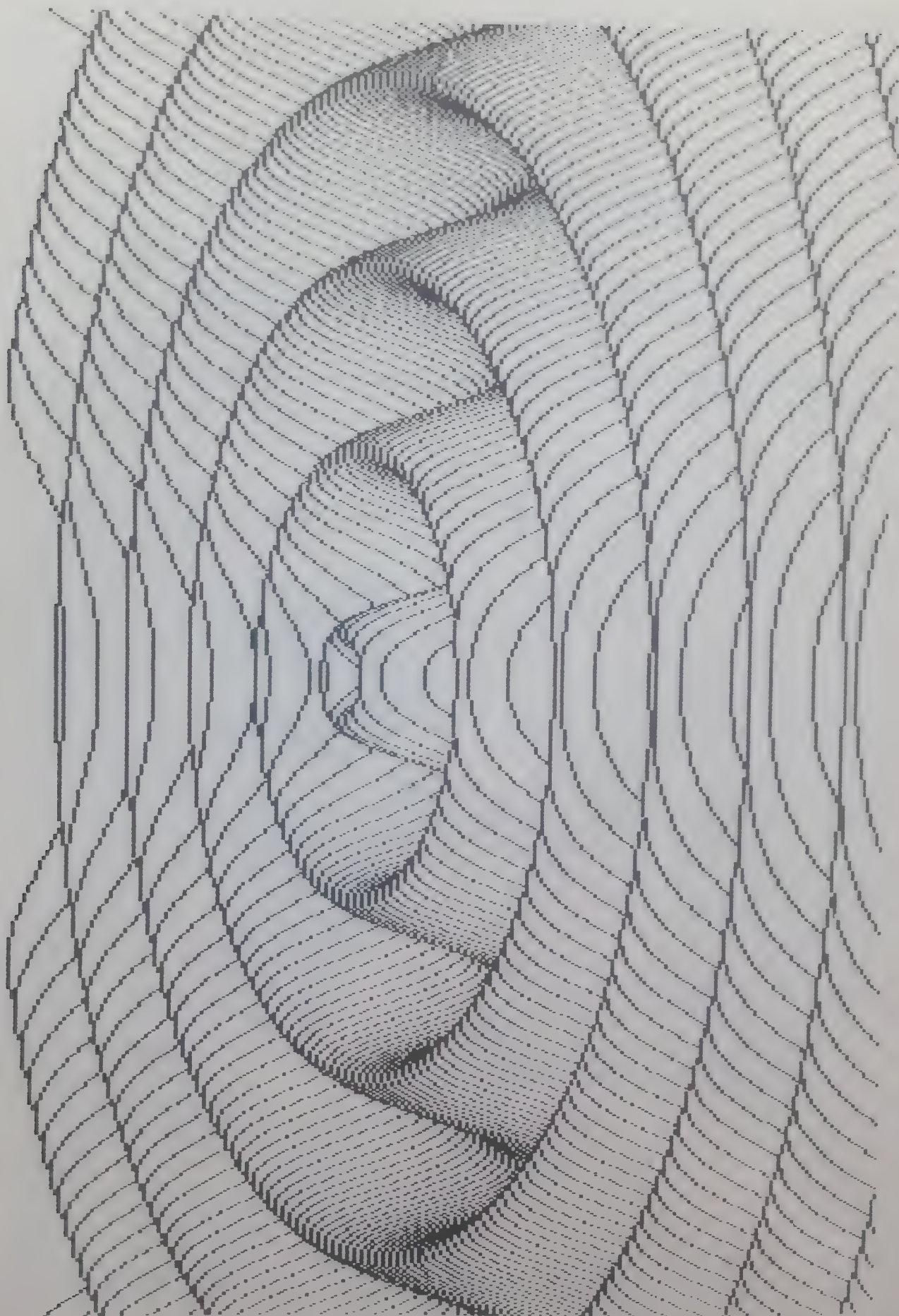
Like the ‘Mexican hat’ program, the function drawn can easily be altered, in this case by changing line 200.

> LIST

```

10 REM Filename:SPLASH
20
30 REM Milk splash
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 VDU 29,640;512;
90 FOR X% = -640 TO 0 STEP 2
100 M% = -512
110 FOR Y% = -1212 TO 912 STEP 16
120 V% = (Y% + FNA(X%,Y%))/2
130 IF M% < V% THEN M% = V%:PLOT 69,X%,V%:PLOT
69,-X%,V%
140 NEXT Y%
150 NEXT X%
160 END
170 DEF FNA(X%,Y%)
180 LOCAL A%
190 A% = SQR(X%*X% + Y%*Y%)
200 = SIN(RAD(A%)*2)*200-A% DIV 5

```

Boolean Arithmetic

An understanding of Boolean algebra is useful for advanced programming techniques in BASIC and assembly language. While leafing through the BBC Micro User Guide you may have seen mention of the AND, OR and NOT operators and so may be aware of what the Boolean operators do.

501p110sl120people only use Boolean operators in IF statements, in the form of IF A=B

AND B< 13 THEN <do something> . This is only one of the many uses of the Boolean operators.

Boolean algebra consists of four basic operations. Three of these take two binary digits and arrive at a third as the answer and the fourth takes a single binary digit and arrives at another binary digit as the answer. Because binary digits can only be 0 or 1, an operator which takes two binary digits as its arguments can only have four possible different inputs. This is because the arguments can only be 0 and 0, 0 and 1, 1 and 0 or 1 and 1.

It is possible to show how each of these operations behave for each possible argument of the operation. The four operations are OR, AND, EOR and NOT. Briefly, the action of each of these operators is as follows. The AND operator takes the two binary digits and if both of them are set to 1, then it returns an answer of 1, otherwise it returns an answer of 0. This is shown in the table below:

INPUT	OUTPUT
00	0
01	0
10	0
11	1

The OR operation returns a value of 1 if either or both of the binary digits are set. Another way of putting it is that it will return 1 unless both the binary digits are zero. This can be shown as follows:

INPUT	OUTPUT
00	0
01	1
10	1
11	1

The EOR operation will return 1 if the two binary digits are different and zero otherwise. This can be shown as:

INPUT	OUTPUT
00	0
01	1
10	1
11	0

The EOR operation means 'exclusive OR', because it is an OR operation that excludes the case when both arguments are 1.

The NOT operation simply inverts on single bit or binary digit. NOT 1 is 0 and NOT 0 is 1. The truth table for NOT is as follows:

INPUT	OUTPUT
0	1
1	0

Before we examine any other uses of these Boolean operators, it is a good idea to see how they can behave as they do when used in IF statements.

The crucial thing is that when the BBC Micro evaluates an IF statement, it is looking for a numeric expression between the word IF and THEN. If this expression evaluates to zero then the expression is said to be FALSE but if it evaluates to a non-zero number, then the expression is said to be TRUE.

Similarly, when you use the comparison operators such as = and >, if the thing they are testing for is TRUE they give a value of a -1 and if the thing they are testing for is FALSE, they return zero. You might expect the value 1 to be used to represent TRUE, but this is not so. The reason -1 is used is because when the BBC Micro does a Boolean calculation, it carries out the operation on each bit of a 32 bit binary number in turn. Thus, 101 AND 110 is 100. TRUE occurs when all the available bits are set to 1 - otherwise NOT TRUE wouldn't be FALSE. When the computer comes to print this string of ones as a decimal number, it first looks at the top bit of the number. This is 1, which indicates (due to the use of 2's complement arithmetic) that the number is negative. As the number is negative, the computer inverts all the bits (getting a string of zeros), then adds one to this answer, getting the answer 1. This is combined with the minus sign to give -1. Simple.

Going back to the original explanation, when you combine two of the comparison operators such as = or >, each gives either TRUE or FALSE. The TRUE and FALSE values are then dealt with quite happily by the AND operator, to give a new value. This new value is the one with which the IF statement deals.

We can also use these operations in other circumstances. The AND operation is

especially useful because it allows us to mask out some of the bits of a number. If you want to `PRINT < a number> AND 7`, the computer will first take the number and then AND it with the binary number 7. The binary number 7 is 111, thus, all the bits of the number will be knocked out except for the bottom 3, since these are the ones 'provided' by the number 7. This can be very useful because the only alternative is to use the MOD operator, as in `PRINT < a number> MOD 8`, which is somewhat inelegant and takes up more time.

If you extract a byte from a graphics screen and EOR it with -1, you will have inverted all the bits in the byte. When the byte is replaced in memory, that area of the screen will be inverted. For example, try running this in MODE 4:

```
FOR T%=&5800 TO &7FFF:?T%=?T% EOR 255:NEXT T%
```

This leads on to another very valuable use of the Boolean operators which has already been thought of by the designers of the BBC Micro.

This is to use the AND, OR, NOT and EOR operators in graphics work to make objects move in front of and behind other objects on the screen, without disturbing anything else on the screen.

Usually the first argument of the GCOL statement is zero, which means that the computer will plot in the colour specified by the next number. Many users of the BBC Micro will ignore what happens when a number other than 0 is used, but the other numbers have quite powerful effects.

To be specific, the options available to the GCOL statement are as follows:

`GCOL 0,< colour>` makes the computer plot in the colour specified.

`GCOL 1,< colour>` makes the computer do a Boolean OR operation between the colour specified and the colour present at any point it is required to plot. Thus,

```
GCOL 1,4
PLOT 69,500,500
```

is equivalent to

```
GCOL 0,4 OR POINT(500,500)
PLOT 69,500,500
```

The reason the second method is not used in preference to the first is that the first method can cope with carrying out these operations throughout a line or a triangle, while the second can only conveniently cope with points.

`GCOL 2,< colour>` is the same, except that the AND operation is used.

`GCOL 3,< colour>` is the same again, except EOR is used.

GCOL 4,< colour> uses the NOT operation, by inverting the colour of every point it has to plot. As NOT only takes a single argument, the value of < colour> is not used in this case.

Those GCOL modifiers allow programs like this to be run:

> LIST

```

10 REM Filename:PRIORY
20
30 REM Graphic priority demo
40
50 REM (c) 1983 Jeremy Ruston
60
70 ON ERROR GOTO 420
80 *FX 11,1
90 *FX 12,1
100 MODE 2
110 FOR X% = 0 TO 39
120 GCOL 0,(X% MOD 5) + 1
130 MOVE X%*32,0
140 DRAW X%*32,1023
150 MOVE X%*32 + 8,0
160 DRAW X%*32 + 8,1023
170 NEXT X%
180 FOR T% = 8 TO 15
190 VDU 19,T%,7;0;
200 NEXT T%
210 VDU 23,224,&3C,&7E,&DB,&DB,&FF,&24,&42,&81
220 X% = 500:Y% = 500
230 VDU 5
240 GCOL 3,8
250 P% = 5
260 L% = 0
270 REPEAT
280 MOVE X%,Y%
290 VDU 224
300 A% = INKEY(10)
310 *FX 15
320 VDU 8,224
330 IF A% = 90 THEN X% = X%-8
340 IF A% = 88 THEN X% = X% + 8
350 IF A% = 47 THEN Y% = Y%-8
360 IF A% = 58 THEN Y% = Y% + 8

```

```

370 IF A% = 44 AND P% > 0 AND L% < > 44 THEN P% = P% - 1:VD
U 19,P% + 9,P% + 1;0;
380 IF A% = 46 AND P% < 5 AND L% < > 46 THEN P% = P% + 1:V
DU 19,P% + 8,7;0;
390 L% = A%
400 UNTIL FALSE
410
420 REPORT
430 PRINT " at line ";ERL
440 *FX 12
450 *FX 12,3
460 END

```

>

When you run this program, a MODE 2 screen will fill up with lots of vertical lines in various colours. Near the middle of the screen will be a little 'space invader' type character. You can use the 'Z', 'X', '/' and '.' keys to move the character left, right, down and up over the screen. No checks are made for the little man exceeding the limits of the screen, so you'll have to be mature while you control it.

You'll notice that the little man moves around the screen OVER the lines, and when he has passed over a particular line, that line is not erased. This in itself is useful, but we can achieve even more interesting effects.

Press the comma key once. From now on, the little man will pass over all the lines - except the purple ones, which he will pass behind. If you press the comma again, he will pass behind the blue lines, and so on until he passes behind all the lines. Once you have reached this point, the comma key stops doing anything. If you press the full stop key, the man starts to pass in front of the red line, and succeeding presses of the full stop key will make him pass in front of progressively more lines, until he passes over them all, as when the program started.

Many commercial games for the BBC Micro use this technique. For example, the 'Monsters' game from Acornsoft uses this technique to ensure that the little men in the game do not erase the various ladders littered around the screen.

This technique is also dependent on the ability to change the logical to physical colour pairings using VDU 19.

To describe the program line by line:

Line 70 sets up an error trap. This is because the program alters the repeat rate and delay of the keyboard, and it is irritating to break out of a program, using Escape, to find that the keyboard is malfunctioning. Thus, the error trap prints the error message, and restores the computer to normal operation.

Lines 80 and 90 set the keyboard to repeat very fast, without a delay before repeating. This is to allow smooth keyboard control.

Line 100 goes into MODE 2. We shall see that although the program only appears to use 6 colours, it in fact uses 13, so MODE 2 is essential. The large number of colours used up is the only disadvantage of this method.

Line 110 sets up a loop to draw the lines.

Line 120 chooses a colour for the line.

Lines 130 to 160 draw two vertical lines next door to each other.

Line 170 terminates the loop.

Lines 180 to 200 sets the colours of logical colours 8 to 15 to be white.

Line 210 defines the little man.

Line 220 sets the starting coordinates of the little man.

Line 230 joins the text and graphics cursors. This is to enable the little man to be drawn anywhere on the screen.

Line 240 sets the plotting action to be 'exclusive OR whatever is on the screen with 8'. This means that each time a point is plotted, the top bit will be toggled. With the colours set up as they are, this ensures that any plotting operation will plot in white.

Line 250 sets the current priority of the little man to be 5. This means he will appear in front of those lines that are colour 5, and all those that precede colour 5 in numerical order.

Line 260 sets the variable that holds the last key pressed to be zero. This is because we have to allow the movement keys to repeat, but not the priority change keys.

Line 270 starts a loop through the main program.

Lines 280 and 290 print the little man at his current position.

Line 300 gets a keypress from the user.

Line 310 clears the keyboard buffer, to ensure that the keyboard does not store extra characters, which could lead to a time lag between what is typed in and what you see on the screen.

Line 320 moves the cursor back a character position, and reprints the little man. The first time the man was printed, whatever was underneath him would have had the top bit set, moving it from a colour in the range 0 to 7 to a colour in the range 8 to 15. When

it is removed in line 320, the opposite happens, so it moves back to be a colour in the range 0 to 7. The crucial point is that all the colours in the range 8 to 15 are white, which makes the whole figure white.

Lines 330 to 360 move the object around by checking for the movement keys.

Line 370 deals with decreasing the priority of the little man. It will only do this if the relevant key is pressed and the priority is greater than zero (a priority of -1 would otherwise result, which would be meaningless). In addition, it makes sure that the last key pressed was not a comma. This makes the priority keys unrepeatable. Once all these conditions are met, the computer decrements the priority. Then it sets the colour of anything of the logical colour corresponding to the original priority be the same as the corresponding colour without the top bit set. This sounds complex, but it effectively means that instead of colour 8+5 (which is the colour generated when the little man is over a purple line) being white, it is purple, giving the illusion that the man is behind the line.

Line 380 reverses this effect to increase the man's priority.

Line 390 sets the last key pressed to be the current key.

To sum the technique up in more general terms:

1. Draw the background / foreground in whatever colours you like, using colours 0 to 7.
2. Set the plotting colour to be GCOL 3,8.
3. If the moving object is to pass in front of a piece of the background / foreground of colour X, then set colour X+8 to be the colour of the object, else set it be the colour of the foreground / background.

Once you have mastered this method, you will see how it can be simplified or otherwise altered to suit your needs.

You might like to write a program like the one in this chapter, except allowing two objects with different priorities to be manipulated.

“Diversion”, “Faster cursor keys”

This practical routine speeds up the action of the cursor and copy keys when they are used in conjunction with the shift key. Without the shift key, they act normally. This makes it easier to alter just a small part of a long line.

The program operates using the ‘key pressed’ event. Whenever a key is pressed, the computer can be instructed to jump to a user written machine code routine. The

routine is also used to 'trap' certain other events, but this facility is not used in this example.

Notice that the machine code of this routine is stored from &A00 onwards. This is not perfect, since it overwrites the cassette/RS423 buffer, but it does prevent the routine from being overwritten if a new program is loaded. The routine could easily be placed at &C00 if user defined characters are not being used.

The program saves the registers as soon as it is entered at line 120. The accumulator contains the number of the event, which is not tested in this example. Line 130 causes the routine to exit if the ASCII code of the key pressed was less than &9B. Line 140 exits if the key number is greater than &9F. This indicates that the key is one of the cursor keys, or the copy key, combined with the shift key.

The key code (without shift) is then inserted into the keyboard buffer using *FX 138. The routine then exits.

The action of the routine ensures that the cursor and copy keys are entered into the buffer twice, once using *FX 138 and once as soon as the routine exits.

Line 220 turns the routine on. You can turn it off using *FX 13,2.

> LIST

```

10 REM Filename:FAST
20
30 REM Speedy edit keys
40
50 REM (c) 1983 Jeremy Ruston
60
70 R% = &A00
80 FOR T% = 0 TO 2 STEP 2
90 P% = R%
100[OPT T%
110.key
120 sta &80:stx &81:sty &82
130 tya:cmp #&9B:bcc exit
140 cmp #&A0:bcs exit
150 sec:sbc #&10:tay
160 lda #138:ldx #0:jsr &FFF4
170.exit
180 lda &80:ldx &81:ldy &82:rts
190]NEXT
```

200 ?&220 = key

210 ?&221 = key DIV 256

220 *FX 14,2

>

Floating Point Arithmetic

Whilst programming in BASIC, you will have met two different types of number: integers and floating point numbers (or reals). Although integers are a subset of reals, the computer has to deal with these two types of numbers in fundamentally different ways. Integers are quite easy to deal with in assembly language. This is because most processors include instructions for the very basic two's complement arithmetic functions.

Real numbers, on the other hand, are much more difficult. One problem is that it is impossible to represent some numbers accurately, as we shall discuss later. So, computers cannot properly be said to carry out real arithmetic - instead they use floating point arithmetic, which is a convenient way to approximate real numbers.

The floating point system is really a variation of the scientific notation used by calculators. The BBC Micro also uses scientific notation for very small and very large numbers. With this system, three pieces of information are required to represent a particular number - as opposed to the two bits required for integers (the sign and the magnitude). The three pieces of information are called the sign, the exponent and the fractional part. We'll first discuss why we need these three parts, then look at how the BBC Micro represents them.

Consider the number:

-5.346E2

In this example, the sign is negative (a missing sign would indicate a positive number), the mantissa is 5.346 and the exponent is 2. The number actually being represented by these pieces of information is obtained by the following expression:

$$-5.346 \times 10^2 = -534.6$$

The number of digits assigned to the mantissa dictates the accuracy with which numbers can be represented, while the number allocated to the exponent dictates how large a range can be covered by a number. For example, if the number of digits allocated to the mantissa is 5, PI would be represented as 3.1416, but with 20 it will be represented as 3.1415926535897932385, which is a much more accurate representation. Similarly, if the number of digits allocated to the exponent is 3, numbers as small as 1E-999 and as large as 1E999 can be represented.

In decimal scientific notation, the number 348.26 would be normally represented as:

3.4826E2

and the number 0.000825 would be normally written as:

8.25E-4

The normal position for the decimal point is to the right of the most significant digit. When it is in that position, the number is said to be in normalised form.

Sometimes, when you do a calculation involving floating point numbers, the answer is unnormalised:

$$(4.1468E2)*(-3.45E-3) = 14.30646E-1$$

The answer can be normalised by shifting the decimal point left or right and adjusting the exponent as we do so. Therefore the above can be normalised by a single shift to the left, yielding:

1.430646E0

Zero is rather special, since there is no way we can normalise it - there is no most significant digit. It can be represented by a mantissa of zero. The exponent is of course arbitrary (in theory). For example, both these numbers are zero:

0.0E345

0.0E-23

The computer does not use scientific notation to base 10, instead it uses binary.

This brings us to a small problem - how can numbers less than one be represented in binary? It turns out that we can extend the binary 'number bar' to the right of the decimal point (called a bicimal point when you use binary arithmetic):

16 8 4 2 1 . 1/2 1/4 1/8 1/16

Thus, the decimal number 5.75 could be represented as the binary number:

101.11

In normalised binary scientific notation, this is:

1.0111E2

In this case, the exponent indicates the power of two by which it must be multiplied.

Strictly speaking, the exponent should have been shown as a binary number, too.

When a binary floating point number is stored in the BBC Micro, one bit is used for the

sign, eight for the exponent and 32 for the mantissa. Since the exponent can be either positive or negative, you might expect it to be stored in 2's complement form. This, however, is not the case. A special constant, called a bias, is added to the 2's complement form. In the case of the BBC Micro, this bias is &80. Thus, an exponent of zero is stored as &80, not &00. In addition, an exponent of zero is used as a special case, for representing the number zero. This is mainly to speed up detection of underflow.

Floating point numbers are always normalised in the BBC Micro. This means that a mantissa of:

00111011

is actually stored as:

111011

The exponent is suitably adjusted. As the number always starts with a binary 1, we don't need to actually store the bit. Instead, it is replaced by the sign bit. Thus, the apparently impossible feat of storing 41 bits of information in 40 bits of storage is achieved.

A floating point number is stored in memory as follows:

XX+0:Exponent

XX+1:MSB of mantissa, with sign bit replacing bit 7

XX+2:Mantissa

XX+3:Mantissa

XX+4:LSB of mantissa

Thus five bytes are required for each number.

We shall now look at the classical methods of doing the four arithmetic operations on floating point numbers. The BBC Micro uses highly optimised forms of these algorithms, which are very difficult to follow unless you know the standard way of doing things.

The first algorithm we shall examine is that for floating point addition. The algorithm forms $U + V$ into the floating point variable W :

1. Separate the exponent and fractional parts of the representations of U and V . This includes moving the sign bit out of the mantissa to where we can get at it and replacing it by a true numeric bit for both numbers.
2. If the exponent of U is less than the exponent of V , swap around U and V . In many cases it is more efficient to combine this step either with the first step or with one of the other steps.
3. Set the exponent of W to be the exponent of U .

4. If the exponent of U minus the exponent of V is greater than or equal to the number of bits in the mantissa, then set the fractional part of W to be the fractional part of U and go on to step 7. This effectively means that if the two numbers are of different orders of magnitude adding the smaller one to the larger one will not make any difference to it, so we can ignore the addition altogether.

5. Shift the fractional part of V to the right by $E_U - E_V$ places, where E_U is the exponent of U and E_V is the exponent of V.

6. Set the fractional part of W to be the fractional part of U plus the fractional part of V. This can be done using a simple integer addition algorithm.

7. Normalise the answer in W. The method of normalising a number is given below.

More informally, the numbers are manipulated until they both have the same exponent (by shifting one mantissa and adjusting one exponent) and then the addition can be carried out fairly normally.

The normalisation algorithm proceeds as follows:

1. Is the most significant bit of the fractional part set? If so, exit, else go on to step 2
2. Shift the fractional part one position to the left
3. Decrement the exponent
4. Go back to step 1

This algorithm will patently not work if you try to normalise zero. The solution? Don't normalise zero - test for it in advance.

Diagrammatically, this makes the floating point number look like this:

XXXXXXXXXX		1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
exponent		mantissa

X indicates a bit which could be set or unset, but it doesn't matter which.

The normalisation process often includes the packing process, whereby the most significant bit is replaced by the sign bit.

To subtract floating point numbers you must negate one of the numbers before using the addition algorithm. This negation can be achieved by simply inverting the status of the sign bit before entering the routine.

Floating point multiplication and division proceed in a very similar manner to each other. Given floating point numbers U and V, they will be multiplied together or U

will be divided by V to gain either the quotient W or the product W. We will first examine the stages for multiplication. (The following descriptions use the notation FW to indicate the fractional part of W, EU for the exponent of U and so on).

The algorithm for multiplication is:

1. Separate the exponent and fraction parts of U and V.
2. Set EW equal to $EU + UV$ and set FW equal to $FU * FV$.
3. Normalise the floating point number W and pack the result, by replacing the most significant bit of FW with the sign bit.

The algorithm for division is as follows:

1. Separate the exponent and fraction parts of the representation of U and V.
2. Set EW equal to $EU - EV$, while FW is set to FU / FV .
3. Normalise the floating point number W.

These descriptions are based on the routines found in the BASIC II ROM. The routines in BASIC I are almost exactly the same, but they are at different places.

If you don't know which version of BASIC your machine has, there is an easy way to find out. Press the 'break' key, and type REPORT. BASIC II will respond with:

(C)1982 Acorn

BASIC I gives the copyright date as 1981.

BASIC has to deal with floating point numbers from a number of sources. To speed things up, all numbers are moved into two sets of page zero locations called the 'floating point accumulators' - I shall abbreviate them to FAC#1 and FAC#2. They are mapped as follows:

SIGN	- &2E (FAC#1)	and	&3B (FAC#2)
OVER/UNDERFLOW	- &2F (FAC#1)	and	&3C (FAC#2)
EXPONENT	- &30 (FAC#1)	and	&3D (FAC#2)
MANTISSA 1	- &31 (FAC#1)	and	&3E (FAC#2)
MANTISSA 2	- &32 (FAC#1)	and	&3F (FAC#2)
MANTISSA 3	- &33 (FAC#1)	and	&40 (FAC#2)
MANTISSA 4	- &34 (FAC#1)	and	&41 (FAC#2)
MANTISSA 5	- &35 (FAC#1)	and	&42 (FAC#2)

(The accumulators are at the same addresses for both versions of BASIC.)

To take each element of the accumulators individually:

The sign byte is used to hold the sign after it has been moved out of the mantissa. Although a full byte is used, only bit 7 is significant. As normal, if bit 7 is set, the number is negative, else it is deemed to be positive.

The over/underflow byte is used to check when accuracy has been lost during a calculation. At the start of a calculation, zero is stored there. Then, every time underflow is detected, it is decremented, and whenever overflow is detected, it is incremented. If, at the end of the calculation, this location still contains zero, no action is taken. If a negative number is found, zero is returned. This ensures that the computer won't do something silly like print 0.000000001 when it means zero, which other BASIC interpreters are prone to do. If the number is found to be positive, the error message 'Too big' is given, and the routine terminates. This scheme allows one underflow to cancel out one overflow and vice versa.

The exponent is entirely standard.

The mantissa is odd, because it has five bytes allocated to it, rather than the four bytes you might expect. The least significant byte (MANTISSA 5) is called the 'rounding byte' in some quarters, and that is exactly the purpose it serves. It helps remove the danger of the situation where a number is shifted to the right and then to the left successively. If the extra byte were not supplied, we would lose a bit for every shift. As a side effect of this, the rounding byte also allows the computer to carry out quite accurate internal calculations - to about 12 significant figures - and then round this figure to the normal precision. Generally, when we speak of the mantissa, we include the rounding byte in the expression.

The mantissa has always had the sign bit in the mantissa replaced by a true numeric bit before it reaches the computer.

We will start by looking at the normalisation algorithm used by BBC BASIC. This description is only in terms of the algorithm used. (For the precise details of how it is implemented, see the ROM listing.)

I will present the algorithms in a sort of bastardised BBC BASIC. They cannot be typed directly into the computer because they use several mythical features, but they should be easy to understand. Particularly esoteric features include the functions B0, B1, B2 etc, which extract the indicated bit from a variable. Notice that some of the time the mantissa is treated as one variable, at others it is an array of the bytes making it up (with the index starting at 1, not zero).

```

10 REM BBC BASIC Normalisation algorithm
20 IF B7(MANTISSA(1)) THEN ENDPROC
30 IF MANTISSA = 0 THEN ENDPROC
40 IF B7(MANTISSA(1)) THEN ENDPROC
50 IF MANTISSA(1)<>0 THEN GOTO 160

```



```
60 MANTISSA(1) = MANTISSA(2)
70 MANTISSA(2) = MANTISSA(3)
80 MANTISSA(3) = MANTISSA(4)
90 MANTISSA(4) = MANTISSA(5)
100 MANTISSA(5) = 0
110 EXPONENT = EXPONENT-8
120 IF EXPONENT didn't dip below zero THEN GOTO 40
130 UNDERFLOW = UNDERFLOW-1
140 GOTO 40
150 IF B7(MANTISSA(1)) THEN ENDPROC
160 Shift whole lot left one position
170 EXPONENT = EXPONENT-1
180 IF EXPONENT didn't dip below zero THEN GOTO 150
190 UNDERFLOW = UNDERFLOW-1
200 GOTO 150
```

Even that may require some comments:

Line 20 leaves the routine if it has been entered with the number already normalised.

Line 30 leaves the routine if the number was zero.

Line 40 again tests to see if the number is normalised. This is the start of a loop, so we have to have another test.

Line 50 decides whether the shift required should be in terms of bits or bytes. If the whole first byte of the mantissa is zero, there will have to be at least one byte shift before we stand a chance of meeting a set bit.

Line 60 starts a byte shift of all the bytes in the mantissa. It acts rather like a chain of firepersons passing buckets. Or, it could be described as being like a row of sports cars edging towards a zebra crossing.

Line 110: we've just done a shift of eight bits, so we need to inform the exponent of this fact. This is done by subtracting eight from it.

Line 120 tests to see if underflow occurred. In the ROM this is done by examining the carry flag, but there is no easy way of showing this in BASIC.

Line 130 adjusts the underflow counter.

Line 140 returns control to the start of the loop. Once there, a further check will be made to see if another byte shift needs to be made. If there isn't, line 50 will pass control to the code to carry out bit shifts.

Line 150 checks to see if the number has yet been normalised.

Line 160 shifts the entire mantissa one position to the left. Again, it was difficult to encode this in BASIC.

Line 170 decrements the exponent to reflect the shift just carried out.

Lines 180 to 200 again check for underflow before passing control back to the start of the loop to check whether the number has been normalised, and whether a new bit shift is needed.

So, it can be seen that the clever thing about the routine is that it doesn't blindly shift the mantissa left - it rather intelligently does the bare minimum number of shifts required. This is because eight shifts next door to each other correspond to a block move

Most of the other routines bear very little resemblance to our original algorithms.

For example, here is the algorithm used for floating point addition. It adds FAC#1 to FAC#2, leaving the answer in FAC#1. Obviously, the same routine is used for subtraction:

(EXP#1 means the exponent of FAC#1, MAN#2 means the mantissa of FAC#2, etc.)

```

10 REM BBC BASIC addition algorithm
20 IF MAN#2=0 THEN ENDPROC
30 IF MAN#1=0 THEN MAN#1=MAN#2:ENDPROC
40 IF EXP#1=EXP#2 THEN GOTO 410
50 IF EXP#1<EXP#2 THEN GOTO 240
60 DIFF=EXP#1-EXP#2
70 IF DIFF>&25 THEN ENDPROC
80 NUM=DIFF AND &38
90 IF NUM=0 THEN GOTO 180
100 NUM=NUM DIV 8
110 FOR TEMP=1 TO NUM
120 MAN#2(5)=MAN#2(4)
130 MAN#2(4)=MAN#2(3)
140 MAN#2(3)=MAN#2(2)
150 MAN#2(2)=MAN#2(1)
160 MAN#2(1)=0
170 NEXT TEMP
180 NUM=DIFF AND 7
190 IF NUM=0 THEN GOTO 410
200 FOR TEMP=1 TO NUM
210 Shift MAN#2 right one bit
220 NEXT TEMP
230 GOTO 410

```



```

240 DIFF = EXP #2-EXP #1
250 IF DIFF > &25 THEN ENDPROC
260 NUM = DIFF AND &38
270 IF NUM = 0 THEN GOTO 360
280 NUM = NUM DIV 8
290 FOR TEMP = 1 TO NUM
300 MAN #1(5) = MAN #1(4)
310 MAN #1(4) = MAN #1(3)
320 MAN #1(3) = MAN #1(2)
330 MAN #1(2) = MAN #1(1)
340 MAN #1(1) = 0
350 NEXT TEMP
360 NUM = DIFF AND 7
370 IF NUM = 0 THEN GOTO 410
380 FOR TEMP = 1 TO NUM
390 Shift MAN #1 right one bit
400 NEXT TEMP
410 IF SIGN #1 = SIGN #2 THEN GOTO 510
420 IF MAN #1 = MAN #2 THEN MAN #1 = 0:ENDPROC
430 IF MAN #1 > MAN #2 THEN GOTO 480
440 MAN #1 = MAN #2-MAN #1
450 SIGN #1 = SIGN #2
460 Normalise
470 ENDPROC
480 MAN #1 = MAN #1-MAN #2
490 Normalise
500 ENDPROC
510 MAN #1 = MAN #1 + MAN #2
520 Normalise
530 ENDPROC

```

(You may have noticed the deliberate error near line 510. In all probability, $\text{MAN \#1} + \text{MAN \#2}$ will overflow to the left. In the ROM whenever this happens the exponent is decremented and the mantissa is shifted one position to the left.)

Line by line notes:

Line 20 checks to see if the second of the two numbers is zero. If it is, the routine simply exits, since the other number is going to be the answer.

Line 30 checks to see if the other number is zero. If it is, it has to copy the second number into the answer accumulator.

Line 40 checks to see if the exponents of the two numbers are already equal. You will

recall that the main purpose of the preamble of the addition routine is to match up the exponents, so if they are matched on entry, the computer takes advantage and skips to the later code.

Line 50 works out which exponent is the smaller. If EXP # 1 is the smaller, then it goes off to perform shifts upon MAN # 1, otherwise it continues and shifts MAN # 2.

Line 60 works out how different the exponents are. This informs us how many shifts are going to be required to do the matching.

Line 70 ignores the addition if the difference is too huge. This is because adding a very small number (like 1) to a very large number (like 34.45E32) will not make a noticeable impression on the larger number.

Line 80 works out how many bytes need to be shifted. If DIFF contains the number of bits, then DIFF DIV 8 is going to contain the number of bytes. This division is not done straight away, since it keeps things nice and fast. Thus, the computer masks all the bits out of the difference, except those that determine the number of bytes to move. The section on Boolean arithmetic should make this use of the AND operation clear.

Line 90 checks to see if there are no bytes to move. If there aren't any, then it skips off to look at how many bits need to be moved.

Line 100 divides the number of bits to move by 8, so that it can be used as a loop index.

Line 110 starts a loop of the number of bytes that need shifting.

Lines 120 to 160 carry out a byte shift to the left (away from the bicimal point).

Line 170 terminates the loop.

Line 180 uses the AND operation again, this time to work out how many bits should be moved.

Line 190 ensures that we don't try to do zero shifts, by skipping the shifting code if it is not needed. The place it skips off to is where the actual addition is done.

Line 200 starts a loop through the number of shifts required.

Line 210 performs the shift itself.

Line 220 terminates the loop.

Line 230 jumps off to the code for doing the actual addition

Lines 240 to 400 are similar to lines 60 to 220, except that they shift MAN # 1.

Line 410 is the start of the code that does the addition itself. First, it checks to see if the two numbers have the same sign. If they have, it goes off to line 510. The point is that if they have the same sign, a normal addition is required, but if they have different signs, it is quicker to assume that they are both positive, but subtract one from the other.

Line 420 checks to see if the mantissas are the same. If they are, the answer will be zero, since the signs are different, and adding X and $-X$ will always yield zero.

Line 430 works out which of the mantissas is the larger. If $MAN\#1$ is larger, then $MAN\#1 - MAN\#2$ must be computed, otherwise $MAN\#2 - MAN\#1$ is computed.

Line 440 does the said subtraction.

Line 450 sets the sign of the answer to be the sign of mantissa number two.

Line 460 normalises the answer, using the normalisation routine we looked at earlier.

Line 470 exits the routine.

Line 480 subtracts the other way, since $MAN\#1$ is the larger.

Line 490 normalises the answer.

Line 500 exits the routine.

Line 510 is used when the numbers have the same sign. It adds the two numbers, using normal integer arithmetic.

Line 520 normalises this answer.

Line 530 exits the routine.

In assembly language, all this is amazingly short.

The routine is faster than the traditional one because it adopts a useful programming principle, the gist of which is 'Never write a single, complex, routine where several simpler ones would do'.

For example, the routine would have been shorter if, instead of having separate code for situations when either exponent is the smaller, simply doing a swap if one exponent is larger. But it would also be slower.

In addition, all the shifting operations use the by now familiar approach of sussing out the number of bytes to shift before thinking about the number of bits.

Another way of looking at it is that most of the instructions in the routine actually do something useful, rather than simply decode material, or move things to the right locations.

Now, for a change, we'll try something difficult. Here is the multiplication algorithm:

(This routine needs to use a third mantissa, called MAN#3. This is located at &42 onwards.)

```

10 REM BBC BASIC Multiplication
20 IF MAN#1 = 0 THEN ENDPROC
30 IF MAN#2 = 0 THEN FAC#1 = 0:ENDPROC
40 TOT = EXP#1 + EXP#2
50 IF TOT didn't go above 255 THEN GOTO 70
60 OVERFLOW = OVERFLOW + 1
70 TOT = TOT - &80
80 EXP#1 = TOT
90 IF TOT didn't dip below zero THEN GOTO 110
100 UNDERFLOW = UNDERFLOW - 1
110 MAN#3 = MAN#1
120 MAN#1 = 0
130 SIGN#1 = SIGN#1 EOR SIGN#2
140 FOR TEMP = 1 TO 32
150 shift FAC#2 right
160 shift FAC#3 left
170 IF a bit fell off the end THEN MAN#1 = MAN#1 + MAN#2
180 NEXT TEMP
190 Normalise
200 ENDPROC

```

Again, this routine fudges the issue of overflow when adding - see line 170. You should also note that UNDERFLOW and OVERFLOW both refer to the same location - they are referred to differently to make the algorithm easier to follow.

Line by line notes:

Line 20 ignores the multiplication request if FAC#1 = 0, since this would obviously give an answer of zero.

Line 30 does the same for FAC#2. Of course, zero has to be copied into the answer accumulator before the routine can be left.

Line 40 adds the two exponents together. Lines 50 and 60 take care of any overflow that may have occurred as a result of the addition.

Line 70 subtracts &80 from the total of the two exponents. This is the bias used by the BBC Micro.

Line 80 uses this answer as the final exponent.

Lines 90 and 100 deal with any underflow generated by this subtraction. This code is a good example of the need for the over/underflow byte. If we simply gave an error message every time the addition at line 40 overflowed, we would be ignoring the fact that the answer might well be reduced to the correct range by subtracting &80 from it.

Line 110 copies MAN#1 to MAN#3. This is because the answer will be built up in MAN#1, but we still need to access the contents of it.

Line 120 places zero in MAN#1. This simply initialises the answer.

Line 130 uses the exclusive OR of the two signs as the sign of the answer. This is because multiplying two numbers of different signs yields a negative answer, while the same signs leads to a positive answer. If you draw a truth table up of these facts, you will see that the EOR instruction takes care of all the considerations.

Line 140 starts up a loop through all 32 bits of MAN#1. This loop is really a normal multiplication routine. So, the whole routine is only a few bits and pieces concerned with getting the sign and exponent right, followed by a nice simple multiplication routine.

Line 150 shifts the multiplicand right. This divides it by two.

Line 160 shifts FAC#3 left. This is simply so we can inspect the least significant bit of it.

Line 170 adds the multiplicand and multiplier if a bit did indeed fall off the end.

Line 180 terminates the loop.

Unfortunately, the method the BBC Micro uses for floating point division is not suitable for conversion into the BASIC-like programs we have previously looked at. Instead, you can look at the precise details in the ROM listing. It turns out that the division algorithm is the one most similar to the classical algorithm.

It is worth looking at some of the algorithms used by BBC BASIC to compute functions and other operators. For example SQR is computed from this algorithm:

(Notice that this algorithm uses extra accumulators. These are indicated as FAC#(X), where X is the address of the fake accumulator).

```

10 REM BBC BASIC square root
20 IF FAC#1=0 THEN ENDPROC
30 IF SGN(FAC#1)=-1 THEN PRINT "Negative root":END
40 FAC#(&46C)=FAC#1
50 EXP#1=(EXP#1 DIV 2)+&40
60 FOR TEMP=1 TO 5
70 FAC#(&471)=FAC#1
80 FAC#1=FAC#(&46C)/FAC#1
90 FAC#1=FAC#1+FAC#(&471)
100 EXP#1=EXP#1-1
110 NEXT TEMP
120 ENDPROC

```

This is a disguised copy of the classical square root algorithm. Rather than try to explain it directly, here is a BASIC version that you can actually RUN normally:

> LIST

```

10 REM Filename:SQRT
20
30 REM SQR demonstration
40
50 REM (c) 1983 Jeremy Ruston
60
70 INPUT "Number:" A
80 PRINT "The machine says:"SQR(A)
90 IF A=0 THEN PRINT "Ruston says:"0:END
100 IF SGN(A)=-1 THEN PRINT "-ve root":END
110 A46C=A
120 EX=TOP+3
130 ?EX=(?EX DIV 2)+&40
140 FOR TEMP=1 TO 5
150 PRINT A
160 A471=A
170 A=A46C/A
180 A=A+A471
190 ?EX=?EX-1
200 NEXT TEMP
210 PRINT "Ruston says:"A

```

>

Running this program will verify that the algorithm works.

Notice that the exponent of FAC#1 (which is stored in the variable A) is accessed by directly messing around in the computers memory. This will only work if A is the first variable to be defined in the program.

“Diversion”, “Bresenham meets Moire”

This program uses a very fast algorithm for drawing circles (Bresenham's algorithm) to draw an intricate display composed of Moire patterns.

The main circle procedure, PROCcircle, only generates the points on the circumference of one 45 degree quadrant of the circle, while PROCcirc__points repeats each point around the circumference of the circle.

In this example, points are not drawn on the edge; lines are inverted from the centre of the screen, through the point in question and off the edge of the screen.

> LIST

```

10 REM Filename:Moire
20
30 REM Moire patterns using Bresenham's circle algorithm
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 XF% = 4:YF% = 4
90 GCOL 4,0
100 PROCcircle(640,512,850)
110 END
120
130 DEF PROCcirc__points(X%,Y%,L%,M%)
140 LOCAL K%
150 K% = 5
160 L% = L%*XF%:X% = X%*XF%
170 M% = M%*YF%:Y% = Y%*YF%
180 MOVE L%,M%:PLOT K%,L% + X%,M% + Y%
190 MOVE L%,M%:PLOT K%,L% + Y%,M% + X%
200 MOVE L%,M%:PLOT K%,L% + Y%,M%-X%
210 MOVE L%,M%:PLOT K%,L% + X%,M%-Y%
220 MOVE L%,M%:PLOT K%,L%-X%,M%-Y%
230 MOVE L%,M%:PLOT K%,L%-Y%,M%-X%
240 MOVE L%,M%:PLOT K%,L%-Y%,M% + X%
250 MOVE L%,M%:PLOT K%,L%-X%,M% + Y%
260 ENDPROC
270
280 DEF PROCcircle(L%,M%,R%)
290 L% = L% DIV XF%
300 M% = M% DIV YF%
310 R% = R% DIV XF%
320 LOCAL X%,Y%,D%
```





```

330 X% = 0
340 Y% = R%
350 D% = 3-2*R%
360 IF X% > Y% THEN GOTO 410
370 PROCcirc__points(X%,Y%,L%,M%)
380 IF D% < 0 THEN D% = D% + 4*X% + 6 ELSE D% = D% + 4*(X%
-Y%) + 10:Y% = Y%-1
390 X% = X% + 1
400 GOTO 360
410 IF X% = Y% THEN PROCcirc__points(X%,Y%,L%,M%)
420 ENDPROC
>

```

“Diversion”, “Blowing up the screen”

This simple program is a good exercise in programming graphics displays in assembly language. It enlarges the top half of the screen so that it fills the entire screen. It only directly works in 20K MODES (0,1 and 2), but it can easily be adapted for all MODEs except MODE 7.

The program in its present form requires you to select a suitable MODE and fill the top of the screen with the required text before you type ‘RUN’.

The first part of the program, from line 120 to 190, builds a table of the start addresses of each horizontal line of pixels. This is done by setting a 16 bit variable to the start address of the screen, &3000, and the Y register to the current row number (counting from the top). The program then loops through all the rows, placing the current address in the table. The address is then incremented. If the current row number is the last in a group of eight, a further constant is added to allow for the organisation of memory. I have coded the problem in a slightly more obscure, but probably faster, way. Executing the immediate mode command ‘MODE 0:FOR T%=&3000 TO &7FFF STEP 4:!T%=-1:NEXT’ will make the memory organization clearer.

The next section, from line 200 to line 230 computes a similar table for the X coordinates. The X coordinates only stretch from 0 to 79 since we are dealing in bytes, not pixels.

Lines 260 to 290 make up the address subroutine. This returns the address of the byte with coordinates X, Y.

Lines 310 to 380 enlarge the screen by taking a pixel from X,Y and replacing it at X,Y*2 and X,Y*2+1.

> LIST

10 REM Filename:ENLRGE

```

20
30 REM Blowing up the screen
40
50 REM (c) 1983 Jeremy Ruston
60
70 DIM R% 2000
80 DIM lsb 256,msb 256,lab 80,mab 80
90 FOR T% = 0 TO 2 STEP 2
100 P% = R%
110[OPT T%
120.construct
130 ldy#0:sty&81:lda#&30:sta&80
140.con1 lda&80:sta msb,Y
150 lda&81:sta lsb,Y
160 inc &81:lda #8:bit &81
170 beq con2:lda &81:eor #128+8:sta&81
180 inc &80:inc &80:lda#128:bit&81:bnecon2:inc &80
190.con2 iny:bne con1
200 ldy#0:sty&80:sty&81
210.con3 lda&80:sta lab,Y:lda&81:sta mab,Y
220 lda#8:clc:adc&80:sta&80:lda#0:adc&81:sta&81
230 iny:cpy#80:bnecon3
240 rts
250
260.address
270 lda lsb,Y:clc:adc lab,X:sta &80
280 lda msb,Y:adc mab,X:sta &81
290 rts
300
310.demo
320 lda#127:sta&82
330.dem1 lda#79:sta&83
340.dem2 ldx&83:ldy&82:jsr address:ldy#0:lda (&80),Y:sta&84
350 ldx&83:lda&82:aslA:tay:jsr address:ldy#0:lda &84:sta (&80),Y
360 ldx&83:lda&82:aslA:tay:iny:jsr address:ldy#0:lda &84:sta (&80),Y
370 dec&83:bpl dem2
380 dec&82:bpl dem1
390 rts
400]NEXT
410 CALL construct
420 REPEAT
430 CALL demo
440 UNTIL GET = 13

```

>

Evaluating Expressions

Most programming languages permit you to write expressions in normal algebraic form, rather than some crazy internal form. For example, you can happily write '2 + 2' in BBC BASIC, and the computer will automatically convert it to its own form.

However, when we start to write our own languages, we have to learn how to do this conversion ourselves.

Most computers convert expressions to reverse Polish notation as their internal form.

There are some other pieces of notation you should be familiar with:

an operand is a passive part of an expression, such as a number or a variable.

an operator is an active part of an expression, such as the symbols $*$, $-$, $+$ and $/$. Operators come in two varieties - binary and unary. Binary operators require two operands to arrive at an answer, while unary operators only require a single operand. Thus, $*$ is a binary operator, whilst the $%$ in ' $a\%$ ' is a unary operator.

In reverse Polish notation, all operators are written after their operands. So we have to write $2\ 2\ +$ rather than $2 + 2$. This doesn't seem particularly helpful, but we shall see later that computers can evaluate reverse Polish expressions rather efficiently and easily.

We can thus reduce our task to that of discovering how to convert expressions to reverse Polish notation.

In fact, BBC BASIC converts expressions to reverse Polish notation so subtly that it is almost transparent. We still need to look at some classical algorithms for converting to reverse Polish notation, since these form the basis of the BBC BASIC method. The compilers we will look at in this book use a fairly classical approach to the conversion.

The expressions we will initially convert will not contain brackets and will only involve the operations $+$, $-$, $*$ and $/$. Operands will be restricted to variables named with a single letter or numbers.

Most of us would give the answer 14 to the sum $2 + 3 * 4$, even though the answer could be interpreted as $(2 + 3) * 4$, which is 20. The reason we arrive at the first answer is that

multiplication is intuitively calculated before addition. Multiplication is said to have a higher precedence than addition or subtraction. The order of precedence of the other operators is usually as follows:

^
*,/
+,-

Thus, '2+3*4' is '2 3 4 * +' in reverse Polish notation, but '(2+3)*4' is '2 3 + 4 *'. Crucially, although we take the precedence of operators into account without too much thought, a computer must be explicitly taught about precedence.

The above example of reverse Polish notation, '2 3 4 * +', may also cause some problems. It could be computed in the following steps:

2 3 4 * +
2 12 +
14

Thus, although the figure 2 appears before the other numbers, it is actually used last.

The way a computer computes reverse Polish notation is rather different. Taking the same example, a computer would scan the expression from left to right.

When it comes across a number (or a variable) it pushes that number onto the stack. If it comes across an operator, it executes that operator using as data the top two numbers on the stack leaving the result on the stack.

At the end of the expression, the answer will be the number on the stack. If more than one number was left on the stack, there were not enough operators in the expression, as in the incomplete expression '2*'.

Unary operators are computed from the single number at the top of the stack.

This BASIC program shows a simple algebraic to RPN routine:

```
> LIST
 10 REM Filename:EXP1
 20
 30 REM RPN converter
 40
 50 REM (c) 1983 Jeremy Ruston
 60
 70 INPUT "Enter expression:" EX$
 80 STACK$ = ""
 90
```

```

100 PROCexp
110 PRINT
120 IF EX$<>"" THEN PRINT "Badly formed expression"
130 END
140
150 DEF PROCpush(A$)
160 STACK$ = STACK$ + A$
170 ENDPROC
180
190 DEF FNpull
200 LOCAL G$
210 G$ = RIGHT$(STACK$,1)
220 STACK$ = LEFT$(STACK$,LEN(STACK$)-1)
230 = G$
240
250 DEF FNprec(A$) = INSTR(" + -*/",A$)
260
270 DEF PROCexp
280 LOCAL TEMP$
290 PROCoperand
300 TEMP$ = EX$
310 PROCoperator
320 IF EX$<>TEMP$ THEN GOTO 290
330 REPEAT
340 IF STACK$<>"" THEN PRINT FNpull;" ";
350 UNTIL STACK$ = ""
360 ENDPROC
370
380 DEF PROCoperand
390 LOCAL L$
400 L$ = LEFT$(EX$,1)
410 EX$ = MID$(EX$,2)
420 IF L$>="A" AND L$<="Z" THEN PRINT L$;" ";;ENDPROC
430 IF L$<"0" OR L$>"9" THEN PRINT "Bad operand":END
440 PRINT L$;
450 L$ = LEFT$(EX$,1)
460 IF L$<"0" OR L$>"9" THEN PRINT " ";;ENDPROC
470 EX$ = MID$(EX$,2)
480 PRINT L$;
490 GOTO 450
500
510 DEF PROCoperator
520 LOCAL L$
530 L$ = LEFT$(EX$,1)
540 IF FNprec(L$)<1 OR L$="" THEN ENDPROC

```



```

550 EX$ = MID$(EX$,2)
560 IF FNprec(L$) <= FNprec(RIGHT$(STACK$,1)) AND STACK$ <
> "" THEN PRINT FNpull;" ";GOTO 560
570 PROCpush(L$)
580 ENDPROC
> RUN
Enter expression:12 + 32
12 32 +
> RUN
Enter expression:12 + 34*45/3
12 34 45 3 / * +
> RUN
Enter expression:1 + 2 + 3 + 4 + 5*6
1 2 + 3 + 4 + 5 6 * +
> RUN
Enter expression:23*23 + 45*45
23 23 * 45 45 * +
>

```

The expression evaluation routine starts off by scanning for an operand. If it can't find one, it gives an error message. Otherwise, the operand is printed out.

Then it checks to see if the next thing in sequence is an operator. If it is not it goes off and unstacks all the operators that are on the stack before exiting.

If it is an operator, it examines the operator to see whether the precedence of the operator on top of the stack is greater than or equal to the precedence of the operator in question. If it is, it unstacks the top operator and goes back to ask itself the question again. If, however, the precedence of the top operator is less than the precedence of the operator in question, it stacks the operator in question and goes back to get another operand.

Unfortunately, the vagaries of BASIC have prevented a readable program being written; particularly irritating in this case is the absence of the WHILE program control statement. It would be a good exercise to try and work out where the WHILE construction could have been used, had it been available.

It is still worth giving line by line notes:

Line 70 gets the expression string from the user. As the expression is scanned, characters are knocked off the left hand side, which removes the need to maintain a separate pointer to the current character in the expression.

Line 80 initialises the thing we'll use as the stack. We can use a single string by simply adding and knocking characters off the right hand end. Luckily, everything we may want to put on the stack is only going to be a single character long, so we won't have too much extraneous code.

Line 100 calls the expression evaluator procedure. The procedure will knock off all the characters it could cope with, leaving the unconverted residue in EX\$.

Line 110 moves the cursor to a new line.

Line 120 checks to see if the entire expression was converted. If it wasn't, it assumes there was an error, and issues the appropriate message.

Line 130 terminates the program.

Lines 150 to 170 comprise PROCpush. They simply add the single character string indicated on to the end of the stack string.

Lines 190 to 230 comprise FNpull. In this case, a character is pulled off the end of the string. Notice the use of a temporary variable.

Line 250 computes the precedence of a given operator, using the INSTR function.

Line 270 starts PROCexp

Line 290 goes to get an operand. PROCoperand will give an error message if it couldn't find one.

Line 300 stores the current state of EX\$ in TEMP\$. This is done so that we can tell if PROCoperator managed to find an operator or not.

Line 310 calls PROCoperator to look for one.

Line 320 checks to see if EX\$ has been changed by PROCoperator. If it has, control is passed back to look for another operand.

Lines 330 to 350 empty the stack, printing everything out as they do so.

Line 380 starts the definition of PROCoperand.

Line 400 extracts the leftmost character.

Line 410 knocks this character off EX\$.

Line 420 checks to see if the character we pulled off was a variable name. If it was, the name is printed, and the routine ends.

Line 430 checks for a digit. If one is not found, an error message is given.

Line 440 prints this first digit.

Lines 450 to 490 repeatedly check for more digits, printing them as they go along.

Lines 510 to 580 comprise PROCoperator.

Line 530 gets the left most character.

Line 540 checks to see if it is a valid operator. If it is not, the routine is exited.

Line 550 extracts this operator from the expression string.

Line 560 checks to see if the precedence of the new operator is less than or equal to that of the one at the top of the stack. If it is, the top operator is pulled and printed.

Line 570 pushes the new operator onto the stack.

Line 580 exits PROCoperator.

There is one problem with this kind of expression evaluator. A simple expression like "2+2" will get converted to the RPN form "2 2 +". This will be computed using the following steps:

1. Load 2 into a register
2. Push the register onto the stack
3. Load 2 into a register
4. Push the register onto the stack
5. Pull the top of the stack to a register
6. Pull the next number to a different register
7. Add the registers
8. Push the result

On the other hand, the only instructions actually needed are:

```
LDA #2
CLC
ADC #2
```

Thus, this method is rather verbose.

We shall now look at an expanded version of the program, which allows such things as brackets and unary operators.

Here is an improved version of the RPN converter program. This time, we can cope with functions (each of which takes a single argument) which are named 'a' to 'z', and the unary plus and minus signs. In addition, exponentiation has been added. Brackets are now legal too.

The sample runs after the listing will give you an idea of how it works.

> LIST

```

10 REM Filename:EXP2
20
30 REM Second RPN converter
40
50 REM (c) 1983 Jeremy Ruston
60
70 REM Variables = A-Z
80 REM Functions = a-z
90 REM Operators = +,*,/,-,^
100
110 STACK$ = ""
120 INPUT "Enter expression:" EX$
130 PROCexp
140 IF EX$<>" " THEN PRINT "Bad operator"
150 IF POS<>0 THEN PRINT
160 END
170
180 DEF FNprec(A$) = INSTR(":- + / * ^ ~ \", A$)
190
200 DEF FNtop = LEFT$(STACK$,1)
210
220 DEF FNPull
230 LOCAL V$
240 V$ = FNtop
250 STACK$ = MID$(STACK$,2)
260 = V$
270
280 DEF PROCpush(A$)
290 STACK$ = A$ + STACK$
300 ENDPROC
310
320 DEF FNcur = LEFT$(EX$,1)
330
340 DEF FNnext
350 LOCAL V$
360 V$ = LEFT$(EX$,1)
370 EX$ = MID$(EX$,2)
380 = V$
390
400 DEF PROCexp
410 LOCAL OP$
420 PROCpush(":")
430 REM Main loop
440 PROCoperand

```

```

450 OP$ = FNoperator
460 IF OP$ = "" THEN GOTO 510
470 IF FNprec(FNtop) >= FNprec(OP$) THEN PRINT FNpull;" ";:
GOTO 470
480 PROCpush(OP$)
490 GOTO 430
500
510 OP$ = FNpull
520 IF OP$ <> ":" THEN PRINT OP$;" ";:GOTO 510
530 ENDPROC
540
550 DEF FNoperator
560 LOCAL OP$
570 OP$ = FNcur
580 IF INSTR(" + * - / ^ ~", OP$) THEN = FNnext
590 = ""
600
610 DEF PROCoperand
620 LOCAL N$, FU$
630 N$ = FNnumber
640 IF N$ <> "" THEN PRINT N$;" ";:ENDPROC
650 N$ = FNvariable
660 IF N$ <> "" THEN PRINT N$;" ";:ENDPROC
670 IF FNcur = "-" THEN PROCpush("~"):N$ = FNnext:GOTO 630
680 IF FNcur = "+" THEN PROCpush("\"):N$ = FNnext:GOTO 630
690 IF FNcur = "(" THEN GOTO 730
700 IF FNcur >="a" AND FNcur <="z" THEN GOTO 740
710 PRINT "Bad operand"
720 END
730 N$ = FNnext:PROCexp:IF FNnext = ")" THEN END
PROC ELSE PRINT "Missing right bracket":END
740 FU$ = FNnext:N$ = FNnext:PROCexp:IF FN
next = ")" THEN PRINT FU$;" ";:ENDPROC ELSE PRINT "Miss
ing right bracket to function":END
750
760 DEF FNnumber
770 LOCAL SA$, SG$
780 SA$ = EX$
790 IF FNcur = "+" OR FNcur = "-" THEN SG$ = FNnext ELSE SG
$ = ""
800 IF FNcur < "0" OR FNcur > "9" THEN GOTO 830
810 SG$ = SG$ + FNnext
820 GOTO 800
830 IF STR$(VAL(SG$)) = SG$ THEN = SG$ ELSE EX$ = SA$: = ""
840

```

```

850 DEF FNvariable
860 IF FNcur> = "A" AND FNcur< = "Z" THEN = FNnext ELSE = ""
>RUN
Enter expression:2 + 2
2 2 +
>RUN
Enter expression:23 + (23 + 23*23)
23 23 23 * + +
>RUN
Enter expression:1 + c(2 + (3-c(4^(3 + -3))))
1 2 3 4 3 -3 + ^ c - + c *
>

```

When the computer has to work out a function, it simply pulls the value off the top of the stack, uses it in the function and places the answer on the stack.

Line by line notes:

Line 110 sets the string variable STACK\$ to the null string. STACK\$ will hold the operator stack.

Line 120 accepts the expression for conversion from the user of the program.

Line 130 calls PROXexp, which will decode as much of EX\$ as it can. When it comes to something it cannot decode, (it could be a comma at the end of a expression - as in the PRINT statement) it will return and EX\$ will contain the portion of the expression it failed to translate. The same idea was used in the previous converter.

Line 140 checks to see if EX\$ is equal to the null string. If it is not, it then prints the message "Bad operator". If EX\$ is completely empty, the entire expression was successfully converted, so no message is given.

Line 150 checks to see if the cursor is positioned at the left edge of the screen, and if it is not, it prints a newline character to move to a new screen line.

Line 180 defines a function which returns the precedence of a given operator. It does this by finding the position of that operator in a master string of operators, which are arranged according to precedence, using the INSTR function. As you can see, the colon is an imaginary operator that I have introduced. It is put onto the stack at each call to PROCexp. It has the lowest precedence of all, which means that it is not pulled off the stack in normal circumstances. When PROCexp has finished, it simply empties the stack until it finds this colon. This allows the procedure to be recursive, as we shall see when we examine how I have implemented brackets. There are two other unusual operators in the list: \ is used to denote a unary plus, whilst ~ is used to denote unary minus.

Line 200 defines FNtop, which returns the operator at the top of the stack without

actually pulling it from the stack. Notice that, for maximum confusion, the stack grows from right to left in this converter, while it worked the other way around in the previous program.

Line 220 starts the definition of `FNpull`, which pulls and returns the top operator from the stack.

Line 230 declares a local variable that we'll need.

Line 240 extracts the top operator and places it in `V$`.

Line 250 shortens the stack string by knocking off the operator we have just placed in `V$`.

Line 260 exits the function with the value `v$`.

Lines 280 to 300 define `PROCpush`, which pushes the operator `A$` on to the stack. It does this in line 290 with the string concatenation.

Line 320 defines `FNcur` which returns the current character of `EX$`. This is done by the `LEFT$` function.

Line 340 starts the definition of `FNnext`. This function returns the next character from `EX$`, like `FNcur`, but it also knocks off the character.

Lines 350 to 380 are similar to `FNpull`, except they extract the character from `EX$`.

Line 400 starts the definition of `PROCexp` which is the main element of the program.

The first step, at line 410, is to declare `OP$` to be a local variable. `OP$` will be held to use the operator under consideration. This is one of the rare occasions where using a local variable is not only pretty, it's necessary. The necessity springs from the recursive nature of the routine.

Line 420 pushes a colon onto the stack. This is the dummy operator used to sense the bottom of the stack.

Line 430 simply marks the start of the program's main loop.

Line 440 calls `PROCoperand` which checks for and prints out the operand at the start of the expression string.

Line 450 sets `OP$` to be the next operator encountered.

Line 460 checks to see if an operator was found. If it wasn't, control is passed on to line 510 which will empty the stack.

Line 470 checks to see if the precedence of the operator on the top of the stack is greater than or equal to the precedence of the operator just found. If so, the top operator is printed and pulled from the stack, and control is passed back to the beginning of the line to try again.

Line 480 pushes the new operator onto the stack.

Line 490 goes back to find the operand that follows the operator.

Lines 510 to 530 pull another operator off the stack, checking if it is the dummy colon. If it was a colon, the procedure is exited, otherwise the operator is printed and return is passed back to line 510 to try again.

Line 550 starts the definition of FNoperator which returns the next operator in the expression string.

Line 570 gets the current character.

Line 580 checks to see if it is a legal operator, using the INSTR operation as before.

Line 590 returns the null string if it was not a valid operator.

Line 610 starts the definition of PROCoperand. This is where it starts to get a bit complicated.

Line 630 gets a number from the left hand side of the expression string.

Line 640 checks to see if a number was found. If it wasn't, N\$ will be null. If the number was found, it is printed out, and the procedure is exited.

Lines 650 and 660 do the same thing for a variable.

Line 670 checks for the presence of a unary minus sign. If one is present, the '~' character is pushed onto the stack to represent the minus sign. Then the minus sign is stripped off with a dummy call to FNnext, after which control is returned to the start of PROCoperand. This is because unary operators are always followed by another operand.

Line 680 does the same for a leading plus sign.

Line 690 checks for left hand bracket, and if one is found, goes off to line 730 to deal with it.

Line 700 checks for a function call, passing control to line 740 if one is found.

Line 710 assumes that a bad operand has been found, and issues the appropriate message.

Line 730 is the section of code dealing with a left hand bracket. First, it pulls the bracket off the expression string, using a dummy call to FNnext. Then it calls PROCexp to evaluate the expression left in EX\$. PROCexp should then return when it comes to the closing bracket. This is checked for, and an error message is issued if it is absent.

Line 740 does the same sort of thing for the brackets in function calls, except that it prints the function name after the function argument has been evaluated.

Lines 760 to 860 are not worth commenting.

We shall be examining the BBC BASIC method when we look at the disassembly and when the SLUG compiler is discussed.

“Diversion”, “Drawing on the tube”

This program draws a complex pattern of a tube with writing on it. The clever point is the way that the writing wraps around the tube. The screen dump shows the output of the program.

The program takes a long time to run, and spends a considerable time building up a SIN and COS table, and a table of the dot patterns of the characters on the tube, so do not be alarmed if the program appears to be malfunctioning.

> LIST

```

10 REM Filename:TUBE
20
30 REM Tubular writing
40
50 REM (c) 1982 Jeremy Ruston
60
70 INPUT "Enter phrase:" A$
80 IF LEN(A$)> 8 THEN A$ = LEFT$(A$,8)
90 IF LEN(A$)< 8 THEN A$ = A$ + " ":GOTO 90
100 LE% = LEN(A$)
110 DIM PO%(LE%,7),X% 100,SI%(35),CO%(35)
120 FOR T% = 0 TO 35
130 SI%(T%) = SIN(RAD(T%*5-45))*250
140 CO%(T%) = COS(RAD(T%*5-45))*250
150 NEXT T%
160 Y% = X% DIV 256
170 FOR T% = 1 TO LE%
180 ?X% = ASC(MID$(A$,T%,1))
190 A% = 10

```




```

200 CALL &FFF1
210 FOR P% = 0 TO 7
220 PO%(T%,P%) = ?(X% + 1 + P%)
230 NEXT P%,T%
240
250 MODE 0
260 VDU 29,640;0;
270 FOR T% = -640 TO 640 STEP 50
280 MOVE T%,1023
290 MOVE T% + 25,1023
300 PLOT 85,T%*6,0
310 PLOT 85,(T% + 25)*6,0
320 NEXT T%
330 VDU 26
340 FOR C% = -400 TO 1100
350 VDU29,C%-(C% MOD 2);C% + (C% MOD 2);
360 PROCsemi(C% + 400)
370 NEXT C%
380*SAVE P.TUBE 3000 8000
390 END
400
410
420 DEF PROCsemi(C%)
430 LOCAL B%,K%,A%,H%
440 B% = 2^(7-(C% DIV 4) MOD 8)
450 K% = 4
460 FOR T% = 0 TO 35
470 H% = (C% DIV 32 + T% DIV 8) MOD LE%
480 IF (PO%(H% + 1,T% MOD 8) AND B%) = 0 THEN GCOL 0,0
ELSE GCOL 0,1
490 PLOT K%,SI%(T%),CO%(T%)
500 K% = 5
510 NEXT T%
520 ENDPROC

```

>

“Diversion”, “Doubling the vertical screen resolution”

This program cannot be guaranteed to work correctly with all monitors - it works with about 50% of those I tried. Colour monitors and televisions work best.

Unless specifically told to do otherwise, the BBC Micro always generates an interlaced picture. Effectively, an interlaced picture means that every scan line is sent to the screen twice very close together, which stops the small black horizontal bars appearing, that sometimes mar the appearance of other computers' output. It should be added that these are pretty unnoticeable, unless you are close to the screen.

This means that in say, MODE 4, the computer is actually sending 512 lines of information, not the 256 you might expect. However, it is possible to gain individual control over these bars which then allows us to have a vertical graphics resolution of 512, rather than the normal value of 256. This would give a resolution of 512 by 320 in MODE 4.

To explain more precisely, every time a vertical sync pulse, which happens every 50th of a second, is sent to the television, the screen will display first odd and then even scan lines. Normally, these two frames are identical, which is exactly what interlace is. On some micros the second set of scan lines is just sent as a black picture. If we can somehow send a different video image at each time when these vertical sync pulses reach it, we should be able to synthesize this extra large vertical resolution which is exactly what this program does. To allow us to control the image sent at each vertical sync pulse, I have decided to use two pages with MODE 4, which means that the screen uses a total of 20K of video RAM.

By switching between these two frames at every vertical sync pulse, it becomes possible to display a different page on the odd and even sync pulses. For this to make sense, one has to place reasonable information in both video frames. This is done by printing some random characters on the MODE 4 screen. Then the odd scan lines are moved to the lower page, and the normal MODE 4 screen has the gaps removed.

> LIST

```

10 REM Filename:DOUBLE
20
30 REM 512 by 320 resolution for MODE 4
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 HIMEM = 12288
90 VDU 22,4
100 PROCassemble
110 PRINT "This is the BBC Micro Compendium.
```

I would like to thank a number of people who either made this book possible, or made it a great deal easier: John Coll; Kathy Goudge, who typed the original";

```

120 PRINT " manuscript; Benedicte de Germay; Chaz Moir and
Rob Pickering from Computer Concepts; Carina Beeson, the ex-
model; The people who make";
```

```

130 PRINT " Southern Comfort; The people who make ginger ale;
Ten Years After; Led Zeppelin; Steppenwolf; Neil Young; Crosby, Sti-
lls and Nash; Joni Mitchell; Jimi Hendrix; Buffalo Springfield; The-
people behind Ribena; Donald Knuth; Dill Tudor";
```



```

140 PRINT "and all the other people too numerous to mention."
150 PROCmove
160
170 REPEAT
180 CALL bot
190 CALL top
200 UNTIL FALSE
210
220 DEF PROCassemble
230 DIM R% 200
240 FOR T% = 0 TO 2 STEP 2
250 P% = R%
260[OPT T%
270.top
280 lda#19:jsr &FFF4
290 lda#12:sta &FE00
300 lda#6:sta &FE01
310 rts
320
330.bot
340 lda#19:jsr &FFF4
350 lda#12:sta &FE00
360 lda#11:sta &FE01
370 rts
380
390]NEXT
400 ENDPROC
410
420 DEF FNhad(X%,Y%) = &5800 + (X%*8) + (Y%MOD8) + (Y%DIV
8)*320
430
440 DEF FNlad(X%,Y%) = &3000 + (X%*8) + (Y%MOD8) + (Y%DIV
8)*320
450
460 DEF PROCmove
470 CALL top
480 FOR Y% = 0 TO 255 STEP 2
490 FOR X% = 0 TO 39
500 A% = ?FNhad(X%,Y%)
510 ?FNlad(X%,Y% DIV 2) = A%
520 NEXT X%,Y%
530 CALL bot
540 FOR Y% = 1 TO 255 STEP 2
550 FOR X% = 0 TO 39
560 A% = ?FNhad(X%,Y%)

```

```
570 ?FNhad(X%,Y% DIV 2) = A%  
580 NEXT X%,Y%  
590 ENDPROC
```

>

To cover the program line by line:

Line 70 puts the computer in MODE 0. Although the program doesn't use MODE 0, it is convenient to use it to clear the top 20K of memory.

Line 80 emphasises that MODE 0 puts HIMEM at 12288 - leaving space for 20K at the top of memory.

Line 90 calls up MODE 4, without altering HIMEM. It does this by accessing the VDU drivers directly.

Line 100 calls the procedure which assembles the machine code used in the program.

Lines 110 to 140 print some random characters on the screen.

Line 150 calls PROCmove, which moves the contents of the MODE 4 screen into the correct format for the new display.

Line 170 starts off the major REPEAT loop of the program.

Line 180 calls the routine BOT, which waits for a vertical sync pulse and then displays the bottom of the two display pages.

Line 190 does the same for the top page.

Line 200 terminates the loop.

Line 220 starts to define PROCassemble.

Line 230 defines some space for the machine code used by the program.

Line 270 starts the assembly ocode for the routine TOP.

Line 280 effects *FX 19.

Line 290 selects the 6845 register 12.

Line 300 places 6 into the register. This makes the address of the screen be $6 \times 256 \times 8$, which is 12288. This corresponds to the lower of the two screens.

Lines 340 to 360 are the same, except that they select the other page.

Line 420 defines the function HAD. This means 'High Address'. It returns the address of a byte with coordinates X,Y on the higher screen.

Line 440 is the same except that it gives the address on the lower screen. It could have been defined as FNhad(X%,Y%)-&2800.

Line 460 starts the definition of PROCmove.

Line 470 calls the TOP routine to ensure that the lower of the two screens is displayed.

Line 480 loops through the spectrum of Y coordinates in steps of two. This makes it step through all the even Y coordinates.

Line 490 steps along the 40 bytes in each line.

Line 500 gets the byte stored at the coordinates in the upper screen.

Line 510 then deposits this byte in the lower screen. Notice that the Y coordinate is divided by two.

Line 520 terminates both loops.

Lines 530 to 580 act in the same way except that they move the odd coordinate bytes up the upper screen.

I should add that the program takes some time to run and once all the characters have been built up, two things can happen. The first is that you can get faced with a rather nasty flickering screen. If this happens you have to press 'escape' to get out of the program and type 'REP. CALL bot:CALL top:U.FA.'. You may have to type this a few times before it works. It makes sense to program this sequence into a function key before you run the program, as this makes it easier to do the typing when the display is switched to the lower screen.

Alternatively, you could have a nice 512 vertical resolution screen with a slightly inordinate amount of flicker. The reason it doesn't always work is that occasionally it tries to display the scan lines in the wrong positions. Thus, although this program is interesting for experimentation, it is not practical due to its inherent unpredictability.

The FROTH threaded language

The language I have chosen to compile in this section is a little like FORTH. FORTH is not quite as useful for general programming as BBC BASIC, but it is extraordinarily easy to compile. The version here is not quite similar enough to FORTH to be called FORTH, so I've taken the liberty of calling it FROTH.

The first thing to bear in mind about any language is that it is based upon a number of standard routines for carrying out tasks such as addition, multiplication, subtraction, division and so on. Before you write a language, it is a good idea to have decided upon the arithmetic routines you are going to use and to have coded these routines. Doing this helps to separate the tasks of writing the compiler, and allows you to spend time ensuring that the routines are as fast as possible.

The routines I would recommend are addition, subtraction, multiplication, division, the modulus operation, the indirection operators, greater than, less than, equal to, not equal to, greater than or equal to, less than or equal to and possibly the Boolean operators AND, OR, NOT and EOR.

Any serious language should have variables that are at least 16 bits wide, which allows it to have a numeric range of either 0 to 65535, or -32768 to 32767 (if two's complement notation is used) which is sometimes more useful for general programming.

Once all the routines have been written, you can start work on the language.

FROTH is based around a stack of 16 bit numbers, like most FORTH implementations. You interact with FROTH by typing one or more words on the keyboard, terminated by pressing carriage return, and separated by spaces. A word can then be made up of any characters except spaces and carriage returns. For convenience, control characters cannot be used either.

The words you type in fall into two categories. If a word is made up of just digits (including a leading plus or minus sign), then the word is converted to a binary number and pushed onto the stack. If it is not a number, the computer searches through a list of known words, and executes any word which matches that typed in. If no match is found, the computer reports an error.

Some examples of these words are '+', '*' and 'PRINT'. '+' pulls two numbers off the stack, adds them together and pushes the answer on the stack, '*' does the same for multiplication and 'PRINT' simply pulls and prints the number on top of the stack.

This simple language description allows the creation of RPN expressions using the other arithmetic words, combined with other words which allow you to define more words, carry out loops and so on.

Here is the listing of FROTH. There are some examples of it in operation following the listing, which will give an idea of how the language works:

> LIST

```

10 REM Filename:FROTH
20
30 REM Hybrid FORTH
40
50 REM (c) 1983 Jeremy Ruston
60
70 MAXCO% = 3000:REM Code size
80 MAXST% = 50*2:REM Stack size
90 MAXDC% = 1000:REM Dictionary size
100
110 PROCinit
120 ON ERROR REPORT:PRINT " at line ";ERL:BUF$ = "":IF E
RR = 17 THEN END ELSE GOTO 140
130 PROClibrary
140 REPEAT
150 PROCprocess
160 UNTIL FALSE
170
180 DEF PROCinit
190 LOCAL A%,B%,C%
200 DIM A% MAXCO%,B% MAXST%,C% MAXDC%
210 AC1% = &50:REM Accumulator
220 AC2% = &52:REM Accumulator
230 AC3% = &54:REM Accumulator
240 STS% = &56:REM Stack start
250 STE% = &58:REM Stack end
260 STP% = &5A:REM Stack pointer
270 !STS% = B%
280 !STE% = B% + MAXST%-1
290 !STP% = B%
300 CDS% = A%:REM Code start
310 CDE% = A% + MAXCO%-1:REM Code end
320 CDP% = A%:REM Code pointer
330 DCS% = C%:REM Dictionary start
340 DCE% = C% + MAXDC%-1:REM Dictionary end
350 DCP% = C%:REM Dictionary pointer

```

```

360 BUF$ = STRING$(255,"*")
370 ENDPROC
380
390 DEF PROCerror(A$)
400 $P% = CHR$(0) + CHR$(100) + A$ + CHR$(0)
410 P% = P% + LEN(A$) + 3
420 ENDPROC
430
440 DEF PROClibrary
450 LOCAL O%,P%,A$,A%
460 FOR O% = 0 TO 2 STEP 2
470 P% = CDP%
480[OPT O%
490
500.PUSH1%
510 LDA AC1%:LDY #0:STA (STP%),Y:INY:LDA AC1% + 1:STA (
STP%),Y
520.aa LDA STP%:CLC:ADC #2:STA STP%:LDA STP% + 1:ADC
#0:STA STP% + 1
530 CMP STE% + 1:BNE bb:LDA STP%:CMP STE%:BNE bb
540]:PROCerror("Stack overflow"):[OPT O%
550.bb RTS
560
570.PUSH2%
580 LDA AC2%:LDY #0:STA (STP%),Y:INY:LDA AC2% + 1:STA
(STP%),Y
590 JMP aa
600
610.PULL1%
620 LDA STP%:CMP STS%:BNE cc:LDA STP% + 1:CMP STS%
+ 1:BNE cc
630]:PROCerror("Stack underflow"):[OPT O%
640.cc LDA STP%:SEC:SBC #2:STA STP%:LDA STP% + 1:SBC
#0:STA STP% + 1
650 LDY#0:LDA (STP%),Y:STA AC1%:INY:LDA (STP%),Y:STA
AC1% + 1:RTS
660
670.PULL2%
680 LDA STP%:CMP STS%:BNE dd:LDA STP% + 1:CMP STS%
+ 1:BNE dd
690]:PROCerror("Stack underflow"):[OPT O%
700.dd LDA STP%:SEC:SBC #2:STA STP%:LDA STP% + 1:SBC
#0:STA STP% + 1
710 LDY#0:LDA (STP%),Y:STA AC2%:INY:LDA (STP%),Y:STA
AC2% + 1:RTS

```



```

720
730.NTRUE%
740 LDA #&FF:STA AC1%:STA AC1% + 1:JMP PUSH1%
750
760.NFALSE%
770 LDA #0:STA AC1%:STA AC1% + 1:JMP PUSH1%
780
790.EQUAL%
800 JSR PULL1%:JSR PULL2%:LDA AC1%:CMP AC2%:BNE NFALSE%:LDA AC1% + 1:
CMP AC2% + 1:BNE NTRUE%:JMP NTRUE%
810
820.NEQUAL%
830 JSR PULL1%:JSR PULL2%:LDA AC1%:CMP AC2%:BNE NTRUE%:LDA AC1% + 1:
CMP AC2% + 1:BNE NFALSE%:JMP NFALSE%
840
850.GREATER%
860 JSR COMPARE%:BMI NTRUE%:JMP NFALSE%
870
880.LESS%
890 JSR COMPARE%:BMI NFALSE%:BEQ NFALSE%:JMP NTRUE%
900
910.LE__EQ%
920 JSR COMPARE%:BPL NFALSE%:JMP NTRUE%
930
940.GR__EQ%
950 JSR COMPARE%:BEQ NTRUE%:BMI NTRUE%:JMP NFALSE%
960
970.COMPARE%
980 JSR PULL1%:JSR PULL2%
990 LDA AC1%:CMP AC2%:BEQ DEQUAL%
1000 LDA AC1% + 1:SBC AC2% + 1:ORA #1:BVS DOVFLOW%:RTS
1010.DEQUAL% LDA AC1% + 1:SBC AC2% + 1:BVS DOV
FLOW%:RTS
1020.DOVFLOW% EOR #&80:ORA #1:RTS
1030
1040.ADD%
1050 JSR PULL1%:JSR PULL2%:LDA AC1%:CLC:ADC AC2%:STA
AC1%
1060 LDA AC1% + 1:ADC AC2% + 1:STA AC1% + 1:JMP PUSH1%
1070
1080.SUBTRACT%
1090 JSR PULL1%:JSR PULL2%:LDA AC2%:SEC:SBC AC1%:STA
AC1%

```

```

1100 LDA AC2% + 1:SBC AC1% + 1:STA AC1% + 1:JMP PUSH1%
1110
1120.OUT%
1130 JSR PULL1%:LDA AC1% + 1:BPL ee
1140 LDA #ASC"-":JSR &FFEE:LDA #0:SEC:SBC AC1%:STA
AC1%:LDA #0:SBC AC1% + 1:STA AC1% + 1
1150.ee LDY #0:.ff LDX #16:LDA #0:.gg ASL (AC1%):ROL
(AC1% + 1):ROL A:CMP #10:BCC hh:SBC #10:INC AC1%:.hh DEX
1160 BNE gg:PHA:INY:LDA AC1% + 1:ORA AC1%:BNE ff:.ii PLA
:CLC:ADC #&30:JSR &FFEE:DEY:BNE ii:RTS
1170
1180.MULT%
1190 JSR PULL1%:JSR PULL2%:LDA #0:STA AC3%:STA AC3% + 1
1200 LDX #16:.jj ASL (AC3%):ROL (AC3% + 1):ASL (AC2%):ROL
(AC2% + 1):BCC kk:LDA AC1%:CLC:ADC AC3%:STA AC3%
1210 LDA AC1% + 1:ADC AC3% + 1:STA AC3% + 1:.kk DEX:BNE
jj:LDA AC3%:STA AC1%:LDA AC3% + 1:STA AC1% + 1:JMP PUSH
1%
1220
1230.NABS%
1240 JSR PULL1%:LDA AC1% + 1:BPL ll:LDA #0:SEC:SBC AC1%:S
TA AC1%:LDA #0:SBC AC1% + 1:STA AC1% + 1:.ll JMP PUSH1%
1250
1260.SDIVMD%
1270 JSR PULL1%:JSR PULL2%:LDA AC2%:STA AC3%:LDA
AC2% + 1:STA AC3% + 1
1280 LDA AC3% + 1:EOR AC1% + 1:STA &90:LDA AC3% + 1:STA
&91
1290 LDA AC1% + 1:BPL CHKDE:LDA #0:SEC:SBC AC1%:STA
AC1%:LDA #0:SBC AC1% + 1:STA AC1% + 1
1300.CHKDE LDA AC3% + 1:BPL DODIV:LDA #0:SEC:SBC
AC3%:STA AC3%:LDA #0:SBC AC3% + 1:STA AC3% + 1
1310.DODIV JSR UDIV:BCS EREXIT:LDA &90:BPL DOREM:LDA
#0:SEC:SBC AC3%:STA AC3%:LDA #0:SBC AC3% + 1:STA
AC3% + 1
1320.DOREM LDA &91:BPL OKEXIT:LDA #0:SEC:SBC
AC3% + 2:STA AC3% + 2:LDA #0:SBC AC3% + 3:STA AC3% + 3:JM
P OKEXIT
1330.EREXIT:]:PROCerror("Division by zero"): [OPT O%
1340.OKEXIT LDA AC3%:STA AC2%:LDA AC3% + 1:STA
AC2% + 1:LDA AC3% + 2:STA AC1%:LDA AC3% + 3:STA
AC1% + 1:RTS
1350
1360.UDIV LDA #0:STA AC3% + 2:STA AC3% + 3
1370 LDA AC1%:ORA AC1% + 1:BNE OKUDIV:SEC:RTS

```

```

1380.OKUDIV LDX #16:.DAVLP ROL (AC3%):ROL (AC3% + 1):RO
L (AC3% + 2):ROL (AC3% + 3)
1390.CHKLT SEC:LDA AC3% + 2:SBC AC1%:TAY:LDA AC3% + 3:S
BC AC1% + 1:BCC DECCNT:STY AC3% + 2:STA AC3% + 3
1400.DECCNT DEX:BNE DAVLP:ROL (AC3%):ROL (AC3% + 1):CL
C:RTS
1410
1420.NDIV%
1430 JSR SDIVMD%:JMP PUSH2%
1440
1450.NMOD%
1460 JSR SDIVMD%:JMP PUSH1%
1470
1480.DUP%
1490 JSR PULL1%:JSR PUSH1%:JMP PUSH1%
1500
1510.SWAP%
1520 JSR PULL1%:JSR PULL2%:JSR PUSH1%:JMP PUSH2%
1530
1540.NDIVMOD%
1550 JSR SDIVMD%:JSR PUSH1%:JMP PUSH2%
1560
1570.NMODDIV%
1580 JSR SDIVMD%:JSR PUSH2%:JMP PUSH1%
1590
1600.DISCARD%
1610 JMP PULL1%
1620
1630.NVDU%
1640 JSR PULL1%:LDA AC1%:JMP &FFEE
1650
1660.NGET%
1670 JSR &FFE0:STA AC1%:LDA #0:STA AC1% + 1:JMP PUSH1%
1680
1690.NAND%
1700 JSR PULL1%:JSR PULL2%:LDA AC1%:AND AC2%:STA
AC1%:LDA AC1% + 1:AND AC2% + 1:STA AC1% + 1:JMP PUSH1%
1710
1720.NOR%
1730 JSR PULL1%:JSR PULL2%:LDA AC1%:ORA AC2%:STA
AC1%:LDA AC1% + 1:ORA AC2% + 1:STA AC1% + 1:JMP PUSH1%
1740
1750.NEOR%
1760 JSR PULL1%:JSR PULL2%:LDA AC1%:EOR AC2%:STA
AC1%:LDA AC1% + 1:EOR AC2% + 1:STA AC1% + 1:JMP PUSH1%

```


1770
 1780.NNOT%
 1790 JSR PULL1%:LDA AC1%:EOR #255:STA AC1%:LDA AC1%
 + 1:EOR #255:STA AC1% + 1:JMP PUSH1%
 1800
 1810 ACE%
 1820 LDA #32:JMP &FFEE
 1830
 1840
 1850 LDA #32:JMP &FFEE
 1860
 1870.CR%
 1880 JMP &FFE7
 1890
 1900 ACES%
 1910 JSR PULL1%:.AA LDA AC1%:ORA AC1% + 1:BEQ BB:LDA #
 32:JSR &F FEE:LDA AC1%:SEC:SBC #1:STA AC1%:LDA AC1% + 1:
 SBC#0:STA AC1% + 1:JMP AA:.BB RTS
 1920
 1930.CHARS%
 1940 JSR PULL1%:JSR PULL2%:.CC LDA AC1%:ORA AC1% + 1:B
 EQ DD:LDA AC2%:JSR &FFEE:LDA AC1%:SEC:SBC #1:STA
 AC1%:LDA AC1% + 1:SBC#0:STA AC1% + 1:JMP CC:.DD RTS
 1950
 1960.NPRINTON%
 1970 LDA #2:JMP &FFEE
 1980
 1990.NPRINTOFF%
 2000 LDA #3:JMP &FFEE
 2010
 2020.NADVAL%
 2030 JSR PULL1%:LDA #128:LDX AC1%:LDY AC1% + 1:JSR &FFF
 4:STY AC1% + 1:STX AC1%:JMP PUSH1%
 2040
 2050.NINKEY%
 2060 JSR PULL1%:LDA #129:LDX AC1%:LDY AC1% + 1:JSR &FFF
 4:STY AC1% + 1:STX AC1%:JMP PUSH1%
 2070
 2080.NREADCH%
 2090 JSR PULL1%:JSR PULL2%:LDX AC2%:LDY AC1%:LDA #135:
 JSR &FFF4:STX AC1%:LDA #0:STA AC1% + 1:JMP PUSH1%
 2100
 2110.INSERT%
 2120 JSR PULL1%:LDA #138:LDX #0:LDY AC1%:JMP &FFF4

```

2130
2140.TV%
2150 JSR PULL1%:JSR PULL2%:LDA #144:LDX AC2%:LDY
AC1%:JMP &FFF4
2160
2170.NTIME%
2180 LDA #1:LDX #&50:LDY #0:JSR &FFF1
2190 JMP PUSH1%
2200
2210.SET__TIME%
2220 JSR PULL1%
2230 LDA #2:LDX #&50:LDY #0:JMP &FFF1
2240
2250]NEXT
2260 CDP% = P%
2270 DATA +,ADD%
2280 DATA -,SUBTRACT%
2290 DATA PRINT,OUT%
2300 DATA *,MULT%
2310 DATA ABS,NABS%
2320 DATA DIV,NDIV%
2330 DATA MOD,NMOD%
2340 DATA DUP,DUP%
2350 DATA SWAP,SWAP%
2360 DATA DIVMOD,NDIVMOD%
2370 DATA MODDIV,NMODDIV%
2380 DATA DISCARD,DISCARD%
2390 DATA VDU,NVDU%
2400 DATA GET,NGET%
2410 DATA AND,NAND%
2420 DATA OR,NOR%
2430 DATA EOR,NEOR%
2440 DATA NOT,NNOT%
2450 DATA SPACE,SPACE%
2460 DATA SP,SP%
2470 DATA CR,CR%
2480 DATA SPACES,SPACES%
2490 DATA CHARS,CHARS%
2500 DATA PRINT__ON,NPRINTON%
2510 DATA PRINT__OFF,NPRINTOFF%
2520 DATA ADVAL,NADVAL%
2530 DATA INKEY,NINKEY%
2540 DATA READCH,NREADCH%
2550 DATA INSERT,INSERT%
2560 DATA TV,TV%

```

```

2570 DATA TRUE,NTRUE%
2580 DATA FALSE,NFALSE%
2590 DATA =,EQUAL%
2600 DATA <>,NEQUAL%
2610 DATA >,GREATER%
2620 DATA < ,LESS%
2630 DATA < =,LE__EQ%
2640 DATA > =,GR__EQ%
2650 DATA TIME,NTIME%
2660 DATA SET__TIME,SET__TIME%
2670 DATA *,0
2680 BUF$ = ""
2690 RESTORE 2270
2700 READ A$,A%
2710 IF A% = 0 THEN GOTO 2790
2720 $DCP% = A$
2730 DCP% = DCP% + LEN($DCP%) + 1
2740 ?DCP% = LEN(A$)
2750 DCP%!1 = A%
2760 DCP% = DCP% + 4
2770 GOTO 2700
2780
2790 TRE% = DCP%
2800 ENDPROC
2810
2820
2830
2840 DEF FNword
2850 LOCAL A$,A%
2860 IF LEN(BUF$) = 0 THEN INPUT LINE "]"BUF$:GOTO 2860
2870 IF LEFT$(BUF$,1) = " " THEN BUF$ = MID$(BUF$,2):GOTO
2860
2880 A$ = BUF$
2890 A% = INSTR(A$," ")
2900 IF A% = 0 THEN BUF$ = "": = A$
2910 BUF$ = MID$(BUF$,A% + 1)
2920 = LEFT$(A$,A%-1)
2930
2940
2950
2960 DEF PROCprocess
2970 LOCAL P%
2980 A$ = FNword:IF LEN(A$) = 0 THEN GOTO 2980
2990 IF FNnum(A$) THEN PROCpush(VAL(A$)):GOTO 2980
3000 P% = FNdick(A$)

```



```

3010 IF P% = 0 THEN PROCcommand(A$):ENDPROC
3020 CALL !(P% + LEN($P%) + 2)
3030 ENDPROC
3040
3050 DEF FN dick(A$)
3060 LOCAL P%
3070 P% = DCS%
3080 IF $P% = A$ THEN = P%
3090 P% = P% + LEN($P%) + 5
3100 IF P% > = DCP% THEN = 0
3110 GOTO 3080
3120
3130 DEF FN num(A$)
3140 = STR$(VAL(A$)) = A$
3150
3160 DEF PROC push(A%)
3170 IF SGN(A%) = -1 THEN A% = 65536 + A%
3180 !AC1% = A%:CALL PUSH1%
3190 ENDPROC
3200
3210 DEF PROCcommand(A$)
3220 IF A$ = "VOCAB" THEN PROCVOCAB:ENDPROC
3230 IF A$ = "BASIC" THEN CLEAR:END
3240 IF A$ = ":" THEN PROCcompile:ENDPROC
3250 PRINT "No such word as ";A$
3260 BUF$ = ""
3270 ENDPROC
3280
3290 DEF PROCVOCAB
3300 LOCAL T%
3310 PRINT "Resident words:"
3320 T% = DCS%
3330 PRINT $T%;" ";
3340 T% = T% + LEN($T%) + 5
3350 IF T% = TRE% THEN PRINT "'User defined words:"
3360 IF T% > = DCP% THEN PRINT:ENDPROC
3370 GOTO 3330
3380
3390 DEF PROCcompile
3400 LOCAL A$,P%,A$,T%
3410 IF DCP% > = DCE% THEN PRINT
"'No dictionary space":BUF$ = "":ENDPROC
3420 A$ = FNword
3430 IF FN dick(A$) > 0 THEN PRINT
"'Word already exists":BUF$ = "":ENDPROC

```

```

3440 $DCP% = A$
3450 DCP% = DCP% + LEN(A$) + 1
3460 ?DCP% = LEN(A$)
3470 DCP%!1 = CDP%
3480 DCP% = DCP% + 4
3490 P% = CDP%
3500 A$ = FNword
3510 IF FNnum(A$) THEN PROCnumber(VAL(A$)):GOTO 3500
3520 IF A$ = ";" THEN [OPT 2:RTS:]:CDP% = P%:ENDPROC
3530 T% = FNtick(A$)
3540 IF T% = 0 THEN PROCimmed(A$):GOTO 3500
3550[OPT 2:JSR !(T% + LEN($T%) + 2):]
3560 IF P% > CDE% THEN PRINT
'"Code too long":BUF$ = "":ENDPROC
3570 GOTO 3500
3580
3590 DEF PROCnumber(A%)
3600 IF SGN(A%) = -1 THEN A% = A% + 65536
3610[OPT 2:LDA #(A% MOD 256):STA AC1%:LDA #(A% DIV 256):
STA AC1% + 1:JSR PUSH1%:]
3620 ENDPROC
3630
3640 DEF PROCimmed(A$)
3650 IF A$ = "BEGIN" THEN PROCbegin:ENDPROC
3660 IF A$ = "END" THEN PROCend:ENDPROC
3670 PRINT '"No such word as ";A$:BUF$ = "":ENDPROC
3680
3690 DEF PROCbegin
3700 !AC1% = P%
3710 CALL PUSH1%
3720 ENDPROC
3730
3740 DEF PROCend
3750 CALL PULL1%
3760[OPT 2
3770 JSR PULL1%:LDA AC1%:ORA AC1% + 1:BNE P% + 5
3780 JMP !AC1%
3790]ENDPROC
> RUN
]VOCAB

```

Resident words:

+ - PRINT * ABS DIV MOD DUP SWAP DIVMOD MODDIV DIS
!CARD VDU GET AND OR EOR NOT SPACE SP CR SPACES CH
ARS PRINT__ON PRINT__OFF ADVAL INKEY READCH INSERT
TV TRUE FALSE = <> > < <= >= TIME SET__TIME

User defined words:

```

]2 2 + PRINT CR
4
]2 2 * PRINT CR
4
]5 4 + 4 * PRINT CR
36
]100 32 DIV PRINT CR
3
]42 43 44 45 46 47 48 49 VDU VDU VDU VDU VDU VDU VDU VDU
10/., + *]
]GET VDU GET VDU GET VDU GET VDU CR
J.R.
]GET GET GET GET VDU VDU VDU VDU CR
.R.J
]45 SPACES 42 VDU CR
                                     *

]42 43 CHARS
*****]
]42 INSERT 2 2
]*
]PRINT CR
4
]
Escape at line 2860
>

```

Before looking at the listing, it makes sense to examine the sample run following the listing.

On typing RUN, FROTH takes a few seconds to initialise, before presenting you with its prompt. In MODE 7 this appears as an arrow. You will not normally have space to run FROTH in other MODEs - it has no graphics statements anyway, so there probably wouldn't be very much point.

The first word I typed was VOCAB. This is like the FORTH word VLIST - it gives a list of all the words that FROTH responds to. There are some exceptions. VOCAB, BEGIN and END are dealt with in an odd way, so they do not appear in the list.

Unlike FORTH, the oldest words appear first. The words are broken into two types - those provided as standard by the system and those added by the user. At start up there are no user defined words, so this section is empty.

The next entry, '2 2 + PRINT CR', is a simple example of FROTH's amazing mathematical capability, showing that FROTH works in RPN. The word PRINT simply pops and prints the top stack item, whilst CR moves the cursor to a new line, since PRINT does not automatically do this at the end of its run. Thus, the above FROTH entry is similar to the BASIC statement:

```
PRINT 2+2
```

The next entry is again similar, except it shows multiplication.

The next entry shows some rather more complex RPN arithmetic - can you work out the algebraic expression that corresponds to the RPN expression featured ?

After a quick division demonstration, the next line shows how the VDU word works. It pops the top item from the stack and sends it direct to the VDU driver.

The point to bear in mind is that this line is equivalent to:

```
VDU 49,48,47,46,45,44,43,42
```

and NOT to:

```
VDU 42,43,44,45,46,47,48,49
```

This is due to FROTH's use of the stack. The next two lines show the same effect. In each case, I typed the letters 'J.R.' after pressing RETURN. In the second example, the letters were transposed due to the action of the stack.

The next two lines show the SPACES and CHARS words. The SPACES word pops a number off the stack, and prints that number of spaces.

CHARS prints X copies of character Y, where X is on top of the stack and Y is under X on the stack.

The final example shows the use of the word INSERT to insert characters into the keyboard buffer.

Notice that the words provided in FROTH are an odd mix. I tried to include one word illustrating each of a number of possible word sources - thus there are the TIME and SET__TIME words to show how OSWORD can be called from FROTH, and the INSERT word to show how OSBYTE can be called.

If the words supplied are unsuitable, the following description will enable you to define your own, more suitable, ones.

Line 70 defines the maximum size of the code area. The code area is used to store the machine code that comprises each word. 3000 is probably a little too generous for most applications.

Line 80 defines the maximum number of entries for the stack. Because the stack is used to store two byte quantities, a stack size of 100 bytes will only allow 50 numbers to be stacked.

Line 90 defines the size of the dictionary. The dictionary is used to store the names and addresses of all the words in the systems. 1000 will allow space for about 100 words.

Line 110 calls PROCinit which simply defines some of the locations used by FROTH.

Line 120 defines the error handling routine. The routine gives the required error message, but only leaves the system if the error was generated by pressing 'escape'. This means that some of the new error messages generated by FROTH (such as 'Stack overflow') will not cause a return to BASIC - which would lose all the user defined words.

Line 130 calls PROClibrary which installs the built in FROTH words.

Line 140 starts the main REPEAT loop of the program.

Line 150 calls PROCprocess, which simply acts upon a single word.

Line 160 terminates this loop by allowing it to continue ad infinitum.

Line 180 starts the definition of PROCinit.

Line 200 reserves space for the code area, the stack and the dictionary.

Line 210 defines the location of the FROTH primary accumulator. These two locations are used for storing numbers during calculations.

Lines 220 and 230 define the secondary and tertiary accumulators.

Line 240 defines the location where the start of the stack is stored. This has to be stored in a page zero location so that the machine code parts of FROTH can detect stack underflow - an attempt to pull a number off an empty stack.

Lines 250 and 260 define the locations for the stack end address and the stack pointer in a similar way to the stack start address.

Lines 270 to 290 set these locations to their default values. Notice that the stack used in FROTH grows upwards - hence the stack pointer is initialised to the start address of the stack. Normal FORTH systems have stacks that grow downwards.

Line 300 sets the default value for CDS% which contains the start of the code area.

Line 310 sets CDE%, which contains the final location of the code area.

Line 320 defines CDP%, which contains the lowest unused location in the code area.

Lines 330 to 350 behave in a similar manner, except that they define the extent of the dictionary area.

Line 360 assigns a whole lot of asterisks to BUF\$. BUF\$ is used to hold the words typed in by the user. It is filled up with asterisks to counteract the odd way in which BBC BASIC assigns string space. Finally, the string is returned to zero length.

Line 370 terminates PROCinit.

Lines 390 to 420 define PROCerror. This procedure is used to insert error messages into assembled code.

Line 440 starts the definition of PROClibrary.

Line 500 starts the code for the push operation. The value held in the primary accumulator is pushed onto the stack.

Line 510 first loads the 6502's accumulator with the low order byte of the number to be pushed, then uses indirect addressing to place this in the stack area. The index register is then incremented before the upper byte is stored.

Line 520 adds two to the current stack pointer value.

Line 530 checks to see if the stack has overflowed over the top of the stack. For this code to operate correctly, the stack must be an even number of bytes long.

Line 540 gives an error if overflow did occur.

Line 550 returns control to the calling routine.

Line 570 starts the definition of the other push routine - this one pushes the contents of the secondary accumulator.

Line 580 stores the accumulator on the stack.

Line 590 passes control back to the other push routine to increment the stack pointer and check for overflow.

Line 610 defines the first pull routine. It pulls the word at the top of the stack to the primary accumulator.

Line 620 checks to see if the stack pointer points to the bottom of the stack. If it does, there is no number on the stack to be pulled.

Line 630 gives an error message if underflow does occur.

Line 640 then decrements the stack pointer by two.

Line 650 gets the top value from the stack, placing it in the primary accumulator.

Lines 670 to 710 are identical to lines 610 to 650, except that they pull the value to the secondary accumulator.

Lines 730 and 740 define the word 'TRUE'. Notice that the label used is prefixed with an 'N' to prevent BASIC tokenising the label. The word TRUE simply places the Boolean value TRUE (or &FFFF) onto the stack. This is achieved by loading the primary accumulator with &FFFF, then pushing it to the stack.

Lines 760 and 770 are similar except that they define the word FALSE, which places the number zero onto the stack.

Lines 790 and 800 define the word EQUAL or '=', which pulls two numbers off the stack, checking whether they are equal. If they are, TRUE is placed on the stack, otherwise FALSE is pushed. This is achieved by pulling the top two values to the primary and secondary accumulators, then comparing the two accumulators byte by byte.

Lines 820 and 830 do the same except that they define the word NEQUAL or '<>', which checks to see if the top two values on the stack are not equal, pushing TRUE if they differ and FALSE if they do not.

Lines 850 and 860 define the word GREATER or '>'. They do this by calling the COMPARE subroutine, which will pull the top two values and bring CMP to bear on them. Depending on the state of the flags when control returns from COMPARE, this routine either branches to NTRUE or NFALSE.

Lines 880 to 890 do the same for LESS or '<'.

Lines 910 and 920 do the same for LE__EQ (less than or equal to), which appears in the vocabulary as '<='.

Lines 940 and 950 do the same for greater then or equal to or '>='.

Lines 970 to 1020 comprise the subroutine COMPARE. This subroutine pulls the top two subroutine values and compares them.

Line 1040 starts to define the word ADD, which appears in the vocabulary as '+'.

Line 1050 pulls the top two values off the stack, then adds their least significant bytes in the normal way.

Line 1060 adds the more significant bytes of the accumulators. Finally, the answer is pushed onto the stack.

Line 1080 starts the definition of the word SUBTRACT or '-'.

Line 1090 pulls the two values from the stack, and then subtracts the less significant bytes of the accumulators.

Line 1100 subtracts the more significant bytes before pushing the answer onto the stack.

Notice that neither the addition nor the subtraction routine bother to check for overflow or underflow of the answer.

Line 1120 starts the definition of the word 'PRINT'. It is labelled OUT to prevent a conflict with the BASIC tokenisation process.

Line 1130 first pulls the value to be printed from the stack into the primary accumulator. It then checks to see if the most significant byte has its top bit set - i.e. whether the number is negative. If it is not, control is passed to the label ee.

Line 1140 first prints the leading minus sign needed for negative values. The value is then negated to make it positive.

Line 1150 starts with the label 'ee', which is where the code for a positive number rejoins that for negative numbers. The rest of the code divides the number and prints it in the normal way.

Lines 1180 to 1210 carries out the multiplication word - again, using the standard shift and add algorithm.

Lines 1230 to 1240 take the absolute value of the number at the top of the stack.

Lines 1260 to 1400 carry out division using the standard shift and trial subtraction method. The division algorithm can be expressed as follows:

1. Check for division by zero
2. Zero accumulator
3. Adjust signs
4. For loop = 16 to 1
5. Shift dividend left into accumulator
6. Subtract divisor from accumulator
7. Increment dividend
8. Goto step 11 if the subtraction didn't generate a carry (in 6502 terms this is a BCS instruction)
9. Add divisor to accumulator
10. Decrement dividend
11. Next loop
12. Remainder is in accumulator
13. Quotient is in dividend

Lines 1420 and 1430 call the division routine to make the word DIV.

Lines 1450 and 1460 call the same routine to make the word MOD.

Lines 1480 and 1490 comprise the word DUP which makes an additional copy of the number on top of the stack.

Lines 1510 and 1520 comprise the word SWAP, which swaps the top two items on the stack.

Lines 1540 and 1550 comprise the word DIVMOD, which carries out a division in the normal way, then places the quotient followed by the remainder on the stack. This is useful if you wish to access both but don't want to call the routine twice for reasons of speed.

Lines 1570 and 1580 are the same, except the remainder is placed on the stack before the quotient.

Lines 1600 and 1610 comprise the word DISCARD which throws away the number on the top of the stack.

Lines 1630 and 1640 comprise the word VDU which passes the byte at the top of the stack to the VDU driver.

Lines 1660 and 1670 comprise the word GET, which places the ASCII value of a keypress on the stack.

Lines 1690 and 1700 comprise the word AND which pulls the top two items off the stack, ANDs them, and then pushes the answer on the stack.

Lines 1720 and 1730 carry out the same processes for the word OR.

Lines 1750 and 1760 carry out the word EOR.

Lines 1780 and 1790 carry out the word NOT, which logically inverts the value at the top of the stack.

Lines 1810 and 1820 comprise the word SPACE, which prints a space on the screen.

Lines 1870 and 1880 comprise the word CR which simply moves the cursor to a new line.

Lines 1900 and 1910 comprise the word SPACES, not to be confused with SPACE, which pulls a number off the top of the stack and prints that number of spaces.

Lines 1930 and 1940 are development of SPACES, called CHARS, which print out multiple repetitions of any character.

Lines 1960 and 1970 comprise PRINT__ON, which turns the printer on using VDU 2.

Lines 1990 and 2000 comprise PRINT__OFF, which turn it off again.

Lines 2020 and 2030 make up the word ADVAL. They pull the channel to be investigated from the stack, leaving the value found on the stack.

Lines 2050 and 2060 make up the word INKEY in an almost identical manner.

Lines 2080 to 2090 make up the word READCH which pulls a pair of coordinates off the stack, and return the character found at their position.

Lines 2110 to 2120 comprise the word INSERT. This word takes the top value off the stack and inserts it into the keyboard buffer, using *FX 138.

Lines 2140 and 2150 make up the word TV which acts exactly like a reversed version of *TV. The only difference is that both parameters must be supplied.

Lines 2170 to 2190 use an OSWORD call to read the current time, and then push this value onto the stack, making up the word TIME.

Lines 2110 to 2230 comprise the word SET__TIME, which sets the clock to the value from the top of the stack.

Line 2260 updates the code pointer by setting it to the address reached so far.

Lines 2270 to 2660 are made up of all the names of the built in words followed by the labels used as aliases. This information is used to build up the dictionary.

Line 2670 marks the end of the data.

Line 2690 restores the data pointer to the start of this table.

Line 2700 reads the first data pair, which is made up of the label followed by its address.

Line 2710 jumps out of this section if the end of the data is encountered.

Line 2720 places the name of the routine in the current dictionary location.

Line 2730 increments the dictionary pointer past the name of the word. Notice that one is added to it to allow for the carriage return following each word.

Line 2740 inserts the length of the word's name at the next dictionary location.

Line 2750 inserts the address of the word at the next two free addresses.

Line 2760 increments the dictionary pointer past the new data.

Line 2770 returns to get a new word.

Line 2790 is where the routine passes control to when it reaches the end of the data list of words. The current dictionary address is used to set the last address used for resident words into the variable 'TRE%'.

Line 2800 exits the routine.

Line 2840 starts the definition of FNword. This function returns a word from the user. It will return the left most word from BUF\$, but if BUF\$ is empty, it will invite the user to enter more words (into BUF\$) before trying again.

Line 2860 checks to see if BUF\$ is empty, and if it is, gets another line of input from the user. Notice the use of INPUT LINE to allow the typing of commas.

Line 2870 strips off any spaces from the left of BUF\$. It checks at this point for BUF\$ being made up of just spaces.

Line 2880 copies BUF\$ into A\$ to prevent a conflict between the value returned and that retained in BUF\$.

Line 2890 looks for the first space in A\$. The letters between the start of A\$ and this space will comprise the first word from A\$.

Line 2900 checks too see if no space was present. If none was, BUF\$ is set to the null string to ensure that an input will be requested next time around, but the routine returns with an unmodified copy of A\$, since this will be the word in question.

Line 2910 removes the first word from BUF\$.

Line 2920 exits the routine with the first word from A\$.

Line 2960 starts PROCprocess which is the main procedure of FROTH.

Line 2980 gets a word from the user.

Line 2990 checks to see if the word is a number. If it is, the number is pushed onto the stack and a new word is accepted.

Line 3000 uses FNdick to get the address in the dictionary of the word typed.

Line 3010 checks to see if the word appeared in the dictionary. If it did not, FROTH goes off to try and interpret it as an immediate command, like VOCAB.

Line 3020 calls the routine if it was found. It does this by finding the start address of the word from the dictionary and then using the CALL command to execute it.

Line 3030 exits the routine.

Line 3050 starts the definition of FNdick - which returns the dictionary address of a given word.

Line 3070 copies the start address of the dictionary to the pointer used in the search.

Line 3080 checks to see if the word defined at the current pointer address is the word being searched for, and if it is, returns with the address pointed to.

Line 3090 increments the pointer onto the next word in the dictionary.

Line 3100 checks to see if the end of the dictionary has been reached - if it has, zero is returned.

Line 3110 then goes back to look again for the word at the new dictionary location.

Line 3130 starts the define FNnum which checks whether its argument is a valid number.

Line 3140 carries out the check by seeing if the string representation of the number represented as a number is the same as the first string representing the number.

Line 3160 starts to define PROCpush which pushes the value of its argument onto the stack.

Line 3170 checks to see if the number is negative - if it is, 65536 is added to it, to bring it into two's complement notation.

Line 3180 inserts the number into the primary accumulator before calling the PUSH routine to place it onto the stack.

Line 3190 exits the routine.

Line 3210 starts to define PROCcommand, which checks to see if the word typed in was a direct command.

Line 3220 compares the word with VOCAB. If there is a match, PROCvocab is called and the routine exits.

Line 3230 checks for BASIC, which simply clears all variables and passes control to BASIC.

Line 3240 calls PROCcompile if the word was a colon. The colon is used to define your own words.

Line 3250 gives an error message if no match was found.

Line 3260 clears the buffer, so that the system stops after finding the error.

Line 3270 exits the routine.

Line 3290 starts to define PROCvocab, which simply prints out the names of all the words in the system.

Line 3310 prints the heading for the first part of the vocabulary.

Line 3320 sets a pointer to the start of the dictionary.

Line 3330 prints the word pointed to, followed by a space.

Line 3340 then increments the pointer onto the next word.

Line 3350 checks to see if the end of the system words has been reached. If it has, the system prints another message.

Line 3360 terminates the routine if the end of the vocabulary has been reached.

Line 3370 goes back to search again.

Line 3390 starts to define PROCcompile, which allows you to define your own words. The practice of defining your own words is described at the end of this description.

Line 3410 ensures that the dictionary is not full. This step is not strictly needed, but it does help to give instant feedback.

Line 3420 gets the next word (following the colon). This word will be the name of the word under definition.

Line 3430 looks for the word in the dictionary. If it exists already, the user is informed, and the attempt to redefine the word is halted.

Line 3440 inserts the name of the word in the dictionary.

Line 3450 increments the dictionary pointer past the name of the word.

Line 3460 places the length of the word into the dictionary space. This information is added to allow backwards searches thorough the dictionary (not presently implemented), which would allow you to redefine words.

Line 3470 inserts the address of the routine into the dictionary space. The address will obviously be the current location in the code area.

Line 3480 increments the dictionary pointer past this new information.

Line 3490 copies the current code location into the pointer P%.

Line 3500 gets the next word. This word will be the first one to actually feature in the definition of the word currently being defined.

Line 3510 checks to see if the word is a number. If it is, PROCnumber is called to assemble the relevant code, and control is passed back to get a new word.

Line 3520 checks to see if a semicolon was entered. If it was, the compilation is ended, so a return is added to the code area. Finally, the code pointer is updated and the routine exits.

Line 3530 searches for the word in the dictionary.

Line 3540 calls PROCimmed to compile so-called immediate key words if the word typed did not appear in the dictionary.

Line 3550 assembles code to execute the word.

Line 3560 checks to see if too much code has been defined.

Line 3570 passes control back to the main routine.

Line 3590 assembles code for a number appearing in a word being compiled.

Line 3600 turns the number into proper two's complement if it is negative.

Line 3610 assembles the relevant code.

Line 3620 exits the routine.

Line 3640 checks for an immediate keyword. This is similar to the piece of code that checked for BASIC and VOCAB in interpretive mode.

Line 3650 checks for the word BEGIN.

Line 3660 checks for the word END.

Line 3670 gives an error message if the word is still unaccounted for.

Line 3690 starts the code to deal with the word BEGIN appearing in the word definition in progress.

Line 3700 loads the primary accumulator with the current address in the code area.

Line 3710 pushes this value onto the stack.

Line 3720 exits the routine.

Line 3740 defines the code for the word END.

Line 3750 pulls the address that BEGIN pushed.

Line 3770 assembles code to pull the value off the top of the stack, and if it is zero, jump back to the address where BEGIN occurred.

Line 3790 exits the routine.

Here are some examples of defining your own words:

> RUN

!VOCAB

Resident words:

+ - PRINT * ABS DIV MOD DUP SWAP DIVMOD MODDIV DISCARD VDU GET AND OR EOR NOT SPACE SP CR SPACES CHAR S PRINT__ON PRINT__OFF ADVAL INKEY READCH INSERT TV TRUE FALSE = <> > < < = > = TIME SET__TIME

User defined words:

]BASIC

>RUN

] : LIFE__THE__UNIVERSE__AND__EVERYTHING 42 PRINT CR ;

]

]LIFE__THE__UNIVERSE__AND__EVERYTHING

42

]THE BBC MICRO

No such word as THE

] : ASTERIX 0 BEGIN 42 VDU 1 + DUP 100 = END ;

]ASTERIX

*****]

] : STARS -1 * BEGIN 42 VDU 1 + DUP 0 = END ;

]STARS

Stack underflow at line 3020

]100 STARS

*****]

] : TYPE__WRITER BEGIN GET DUP VDU 13 = END CR ;

]TYPE__WRITER

This is FROTH acting as a typewriter

]VOCAB

Resident words:

+ - PRINT * ABS DIV MOD DUP SWAP DIVMOD MODDIV DISCARD

D VDU GET AND OR EOR NOT SPACE SP CR SPACES CHAR S PRINT

NT__ON PRINT__OFF ADVAL INKEY READCH INSERT TV TRUE FALSE = <> > < < = > = TIME SET__TIME

User defined words:

LIFE__THE__UNIVERSE__AND__EVERYTHING ASTERIX STARS TYPE

E__WRITER

]

The above dialogue also shows some of the other features of FROTH that we have so far neglected.

You should be able to see how words are defined. The format is a colon, followed by the name of the word you wish to define, followed by the words that comprise the word under definition. The definition is terminated with a semi colon.

A user defined word is executed in the same way as any other.

The above text also shows the BEGIN-END construction. This can only be used when you are defining words. When the word END is encountered, control will be passed back to the word BEGIN only if the number at the top of the stack is zero. You can see how I have used this construction to make up loops.

To summarise all the words implemented:

+	adds the top two numbers on the stack, pushing the answer
-	subtracts the number at the top of the stack from the number immediately below it, leaving the answer on the stack
PRINT	pops the number on the top of the stack, and prints it in decimal
*	multiplies the two numbers on the top of the stack, leaving the result on the stack
ABS	pops the number on the top of the stack, takes its absolute value, and pushes the result
DIV	divides the second item on the stack by the top item, leaving the result on the stack
MOD	divides the second item on the stack by the top item, leaving the remainder on the stack
DUP	duplicates the item on the top of the stack, by making two copies of it
SWAP	swaps the top two stack items
DIVMOD	divides the second item on the stack by the first, leaving the quotient and the remainder on the stack
MODDIV	leaves the quotient and the remainder the other way around
DISCARD	discards the number at the top of the stack
VDU	pulls the top number from the stack, and sends it to the VDU driver
GET	waits for a keypress, then places the ASCII code of the key on the top of the stack
AND	Boolean AND between top two numbers on the stack
OR	Boolean OR
EOR	Boolean EOR
NOT	takes the one's complement of the number on the top of the stack
SPACE	prints a space
SP	prints a space

CR	moves the cursor to a new line
SPACES	pops a number of the stack, and prints that number of spaces
CHARS	pops two numbers off the stack. The second number is passed to the VDU driver the number of times indicated by the top number
PRINT__ON	engages the printer
PRINT__OFF	disengages the printer
ADVAL	pulls the channel number of the top of the stack, then pushes the current conversion value for that channel
INKEY	pulls the time limit off the top of the stack, then performs an INKEY operation, before pushing the answer back on the stack
READCH	pulls the X and Y coordinates of a character cell from the stack, then pushes the ASCII code of the character found at the indicated position
INSERT	pulls a number from the top of the stack, and places it in the keyboard buffer
TV	mimics the *TV command
TRUE	pushes the value TRUE, or -1, onto the stack
FALSE	pushes the value FALSE, or 0, onto the stack
=,<>,<,>,<=,>=	standard comparison operators
TIME	pushes the current TIME onto the stack
SET__TIME	pops a number off the stack, and uses it to set the clock

“Diversion,” “Screen Oddity”

This program dramatically shows the effect of updating the screen faster than the screen is displayed. It is instructive to press a key while it is running. The only way out of the program is to press 'break'.

> LIST

```

10 REM Filename:SLOW
20
30 REM Screen oddity
40
50 REM (c) 1983 Jeremy Ruston
60
70 DIM R% 100
80 S% = &7C00
90 FOR T% = 0 TO 2 STEP 2
100 P% = R%
110[OPT T%
120.clear

```



```
130 lda #255
140 jsr empty
150 lda #32
160 jsr empty
170 jmp clear
180
190.empty
200 ldy #0
210.em1 sta S%,Y
220 sta S% + 256,Y
230 sta S% + 512,Y
240 sta S% + 768,Y
250 iny:bne em1
260 rts
270]NEXT
280 MODE 7
290 CALL clear
```

>

The SLUG structured language

This section describes a compiler for a new language called SLUG.

SLUG is half-heartedly meant to stand for 'Structured Language with Universal Greatness'. It is rather like Pascal in concept, but can be persuaded to look sufficiently like BASIC for BBC BASIC programmers to use it easily.

The compiler itself generates code that is between 10 and 60 times faster than an equivalent BASIC program. The code it generates comes out in the form of assembly language statements. The assembly language must be read back into the computer for conversion to machine code. This means that cassette users will not realistically be able to use the compiler, although all readers should gain something from it.

Programs are only compiled to assembly language (rather than machine code) to allow you to edit the object program, perhaps to speed up crucial parts. It also allows the compiler to be simpler.

The program you wish to compile must be in a normal ASCII text file on disc. There are several ways you can generate it. If you have access to a wordprocessor such as Wordwise, you can write the program directly. If you use a different wordprocessor, you must check to ensure that the only non-ASCII character it places in text is the carriage return code. If you don't have a wordprocessor, you can use the BUILD command of the Acorn DFS, although this command does not allow you to edit text files easily. A third method is to use a simple BASIC program. Such a program could be:

> LIST

```
10 REM Filename:EDIT
20
30 REM Simple text editor
40
50 REM (c) 1983 Jeremy Ruston
60
70 DATA ":This is the text of the program"
80 DATA "BEGIN"
90 DATA "VAR CHARACTER"
100 DATA "CHARACTER = 32"
```

```

110 DATA "WHILE CHARACTER< 127"
120 DATA "BEGIN"
130 DATA "OUTPUT CHARACTER"
140 DATA "CHARACTER = CHARACTER + 1"
150 DATA "END"
160 DATA "END"
170 DATA *****
180 INPUT "Filename:" FILE$
190 OT% = OPENOUT(FILE$)
200 IF OT% = 0 THEN PRINT "Unable to open file":END
210 READ A$
220 IF A$ = "*****" THEN CLOSE#0:END
230 FOR T% = 1 TO LEN(A$)
240 BPUT#OT%,ASC(MID$(A$,T%))
250 NEXT T%
260 BPUT#OT%,13
270 GOTO 210

```

>

You can enter the program to be run in the DATA statements. The normal BASIC editor will allow you to edit the program. (Don't worry about the program in the above DATA statements for the moment).

Once you have created a source file, it can be read into the compiler. The compiler will only scan through it once, looking at it a line at a time. Because the entire source program does not have to be held in memory at the same time, the compiler itself can be quite long.

As it scans through it, it generates a BASIC program, which it pumps to a specified file. The BASIC program contains all the assembly language statements corresponding to the original program.

To run the compiled program, you must *EXEC the object program. This is shown in a sample run later.

Before starting to use the compiler in earnest, I'll describe the language SLUG in some detail. Then I will present some sample programs, followed by the compiler listing and comments on the way the compiler works.

The syntax of SLUG can best be explained using the notation detailed on page 198 of the User Guide.

A SLUG program is defined as follows:

< program> ::= < statement>

This means that a program can only consist of a single statement. At first sight, this seems a little restricting. However, there is a special kind of statement available, called a compound statement, which permits you to write many statements where a single statement is permitted.

The various kinds of statements available are as follows:

>>>>> THE COMPOUND STATEMENT

The compound statement consists of the word **BEGIN** on one line, followed by any number of other statements (all on separate lines), terminated by the word **'END'**. This compound statement is treated exactly like a normal, single line statement. You can nest compound statements as much as you wish.

BEGIN

{ < statement> }

END

EXAMPLE:

BEGIN

VAR H

H = 12

WRITE H'

END

>>>>> THE COMMENT STATEMENT

Any line beginning with a colon (once any spaces at the start of the line have been ignored) is assumed to be a comment line. All comments are reproduced in the object code file.

EXAMPLE:

:This is a comment

>>>>> THE VAR STATEMENT

The **VAR** statement is used to declare variables. SLUG has to know the names of all the variables you wish to use in advance. Variables are 16 bits wide, giving them an approximate range of -32700 to 32700.

VAR < var__name> { ,< var__name> }

Long variable names do not slow SLUG down, as they do BASIC. It makes sense to use quite meaningful names.

The maximum number of variables you can use in a program is 35.

Variable names must follow the same restrictions as normal BBC BASIC programs, with the added constraint that the following cannot be used:

FALS

AC1

IAC2

TRU

EQUAL

NO__EQUAL

COMPARE

DEQ

DOV

GREATER

LESS

LESS__EQUAL

GREATER__EQUAL

MULTIPLY

JJ

KK

PNTNUM

EE

GG

HH

II

FF

PNTSTNG

ZA

ZB

ZC

EXAMPLE:

VAR H,J

>>>>> THE IF STATEMENT

The IF statement resembles the IF statement in BASIC. The syntax is:

IF < exp> THEN

< statement>

[ELSE

< statement>]

Notice that the statement is spread over a number of lines. The ELSE section is optional.

< exp> represents an expression - the format of expressions will be explained in due course.

If this expression evaluates to TRUE (defined to be anything but zero), the statement following the word THEN is executed. If it is FALSE (zero), the statement after ELSE (if present) will be executed.

EXAMPLE:

```
IF A = 12 THEN
OUTPUT "Yes"
ELSE
OUTPUT "No"
```

>>>>> THE INDIRECTION OPERATORS

The indirection operators are used in a similar way to those of BBC BASIC. The section on expressions describes how to use them for interrogating addresses - here, we are only concerned with the way they can be used to alter addresses.

Unlike BBC BASIC, only one argument can be present. Thus, you can write

```
!A = B
```

But not

```
A!3 = B
```

In a compiled language such as SLUG, writing $!(A + 3)$ is just as fast.

The word indirection operator only acts upon two byte words in the normal 6502 order - low byte followed by high byte.

EXAMPLE:

```
!B = 12
?(D + 3) = CHARACTER
```

>>>>> THE WHILE STATEMENT

This statement is similar to the BBC BASIC REPEAT-UNTIL construction. The syntax is:

```
WHILE < exp >
< statement >
```

When the WHILE statement is encountered, the expression $\langle \text{exp} \rangle$ is evaluated. If it is FALSE the next statement is skipped. If it is TRUE, the next statement is executed. After execution, the test is made again.

Thus, unlike the REPEAT-UNTIL construction, the condition is tested at the start of the loop, not the end. This means that the $\langle \text{statement} \rangle$ might not be executed at all.

EXAMPLE:

```
WHILE A < 12  
A = A + 1
```

>>>>> THE UNTIL STATEMENT

This statement takes the form:

```
UNTIL < exp >  
< statement >
```

It is exactly the same as the BBC BASIC REPEAT-UNTIL construction. Notice that the expression is written before the statement, even though the statement is executed before the expression is evaluated.

EXAMPLE:

```
UNTIL A = 32767  
A = A + 1
```

>>>>> THE EXIT STATEMENT

This statement is used to leave a program prematurely. If a program only exits when it reaches its physical end, the EXIT statement is not needed.

The syntax is simple:

```
EXIT
```

>>>>> THE OUTPUT STATEMENT

This statement is identical to the BBC BASIC statement VDU.

The same use may be made of commas and semi-colons.

EXAMPLE:

```
OUTPUT 23;8202;0;0;0;
```

>>>>> THE INC & DEC STATEMENTS

These statements respectively increment and decrement a variable (by 1). The syntax used is:

INC < var__name>

or

DEC < var__name>

EXAMPLE:

DEC H

INC JK

>>>>> THE WRITE STATEMENT

This statement is almost identical to the PRINT statement of BASIC.

The only difference is that if you want the cursor to end up on a new line, you must explicitly include a tick (').

The components of the statement are:

TAB(< exp>) which moves the cursor to the specified column

TAB(< exp> ,< exp>) which moves the cursor to a specified position

; which does nothing

, which prints a space

In addition, the WRITE statement can include quoted strings and expressions, which are treated in the normal way.

EXAMPLE:

WRITE "The answer is ";ANS

>>>>> THE ASM STATEMENT

This statement allows you to mix assembly language with your SLUG program. All lines between the word ASM and a line containing ENDASM are passed directly to the object code file.

EXAMPLE:

ASM

LDA #19:JSR &FFF4

ENDASM

>>>>> EXPRESSIONS

The structure of expressions is almost identical to that used in BBC BASIC. The following binary operators are used:

OR
EOR
AND
=
<>
> =
< =
>
<
+
-

This list is in reverse order of precedence.

The unary operators allowed are:

-
+
?
!

In addition, brackets can be used in the normal way.

The only pseudo variable allowed is GET, which returns a keypress.

You'll notice that there are several glaring omissions from the above specification. Graphics are not supported, and words like INKEY are not present. They are very easy to add after you have read the program description later in this section.

SLUG as it stands is really the bare minimum that makes an acceptable language. If you have a special application, the onus is on you to extend SLUG.

A word of warning - if you are thinking of adding graphics extensions, remember that as soon as you use the operating system routines, you will effectively lose all the speed gained over BASIC by compiling.

This is because you will be using the same routines (those in the VDU driver) that BASIC uses.

Here is a typical SLUG program:

```

:A demonstration of looping
BEGIN
  VAR FRED
  FRED = 0
  UNTIL FRED = 32767
    INC FRED
END

```

First, notice the use of indentation (added spaces at the start of each line) to aid readability. It allows you to see quite easily the extent of each loop and compound statement.

The program simply increments the variable FRED 32768 times. Thus, it corresponds to the BASIC fragment:

```
FRED=0
REPEAT
  FRED=FRED+1
UNTIL FRED=32767
```

Here is the dialogue with SLUG used to compile and then execute the program:
(The lines starting with two asterisks are my comments.)

```
** First, the compiler is loaded from disc
>LOAD "SLUG"
** Out of interest, I noted that it was 13.75K long
>PRINT ~TOP-PAGE
      36E5
>P.(TOP-PAGE)/1024
13.7236328
** The compiler was then run. You cannot see it here, but it automatically
puts the computer in MODE 7
>RUN
```

Jeremy Ruston's

Structured
Language with
Universal
Greatness

** These filenames follow a useful convention of using the directory of files to indicate whether they are source or object.

The file S.LOOP had been previously prepared

Enter source filename:S.LOOP

Enter object filename:O.LOOP

** The compiler now prints out the object code as it is generated

```
10 HIMEM = TOP + 500
```

```
20 IAC1 = &50
```

```
30 IAC2 = &52
```

```
40 FOR T% = 0 TO 2 STEP 2
```

```
50 P% = HIMEM
```

```
60 [OPT T%
```

** The object code starts with a list of routines that are used by some commands. Only the EQUAL routine is used in this case.

If you were short of space, you could delete any unused routines.

```
70 /Run time library
```

```
80 .FALS LDA #0:STA IAC1:STA IAC1+1:RTS
```

```
90 .TRU LDA #&FF:STA IAC1:STA IAC1+1:RTS
```

```

100 .EQUAL LDA IAC1:CMP IAC2:BNE FALS:LDA IAC1 + 1:CMP I-
AC2 + 1:BNE FALS:BEQ TRU
110 .NO_EQUAL LDA IAC1:CMP IAC2:BNE TRU:LDA IAC1 + 1:C-
MP IAC2 + 1:BNE TRU:BEQ FALS
120 .COMPARE LDA IAC2:CMP IAC1:BEQ DEQ
130 LDA IAC2 + 1:SBC IAC1 + 1:ORA #1:BVS DOV:RTS
140 .DEQ LDA IAC2 + 1:SBC IAC1 + 1:BVS DOV:RTS
150 .DOV EOR #&80:ORA #1:RTS
160 .GREATER JSR COMPARE:BMI TRU:BPL FALS
170 .LESS JSR COMPARE:BMI FALS:BEQ FALS:BNE TRU
180 .LESS_EQUAL JSR COMPARE:BPL FALS:BMI TRU
190 .GREATER_EQUAL JSR COMPARE:BEQ TRU:BMI TRU:BPL
FALS
200 .MULTIPLY
210 LDA #0:STA &31:STA &32
220 LDX #16:JJ ASL &31:ROL &32:ASL IAC2:ROL IAC2 + 1:BCC KK
:LDA IAC1:CLC:ADC &31:STA &31
230 LDA IAC1 + 1:ADC &32:STA &32:KK DEX:BNE JJ:LDA &31:STA
IAC1:LDA &32:STA IAC1 + 1:RTS
240 .PNTNUM LDA IAC1 + 1:BPL EE:LDA #ASC"-":JSR &FFEE:LDA
#0:SEC:SBC IAC1:STA IAC1:LDA #0:SBC IAC1 + 1:STA IAC1 + 1
250 .EE LDY #0:FF LDX #16:LDA #0:GG ASL IAC1:ROL IAC1 + 1
:ROL A:CMP #10:BCC HH:SBC #10:INC IAC1:HH DEX
260 BNE GG:PHA:INY:LDA IAC1 + 1:ORA IAC1:BNE FF:II PLA:CLC
:ADC #&30:JSR &FFEE:DEY:BNE II:RTS
270 .PNTSTNG PLA:STA IAC1:PLA:STA IAC1 + 1:LDY #0:ZA INC
IAC1:BNE ZB:INC IAC1 + 1:ZB LDA (IAC1),Y:BEQ ZC:JSR &FFEE:
JMP ZA:ZC LDA IAC1 + 1:PHA:LDA IAC1:PHA:RTS
280 /End of run time library
290 .START%
300 /A demonstration of looping
310 ]
320 FRED = &54
330 [OPT T%
340 LDA #0 MOD 256:STA FRED:LDA #0 DIV 256:STA FRED + 1
350 .L000
360 INC FRED:BNE L001:INC FRED + 1:L001
370 LDA FRED:STA IAC2:LDA FRED + 1:STA IAC2 + 1
380 LDA #32767 MOD 256:STA IAC1:LDA #32767 DIV 256:STA IAC
1 + 1
390 JSR EQUAL
400 LDA IAC1:ORA IAC1 + 1
410 ]:IF (P%-L000)< = 126 THEN [OPT T%:BEQ L000:] ELSE [OPT
T%:BNE L003:JMP L000:L003:]
420 [OPT T%

```

430 RTS

440]NEXT T%

450 CALL START%

** Before the object code can be run, the compiler has to be cleared out of memory

>NEW

** The *EXEC command then reads in the object file as a list of BASIC commands.

>*EXEC O.LOOP

>10 HIMEM = TOP + 500

>20 IAC1 = &50

>30 IAC2 = &52

>40 FOR T% = 0 TO 2 STEP 2

>50 P% = HIMEM

>60 [OPT T%

>70 /Run time library

>80 .FALS LDA #0:STA IAC1:STA IAC1 + 1:RTS

>90 .TRU LDA #&FF:STA IAC1:STA IAC1 + 1:RTS

>100 .EQUAL LDA IAC1:CMP IAC2:BNE FALS:LDA IAC1 + 1:CMP IAC2 + 1:BNE FALS:BEQ TRU

>110 .NO__EQUAL LDA IAC1:CMP IAC2:BNE TRU:LDA IAC1 + 1:CMP IAC2 + 1:BNE TRU:BEQ FALS

>120 .COMPARE LDA IAC1:CMP IAC2:BEQ DEQ

>130 LDA IAC1 + 1:SBC IAC2 + 1:ORA #1:BVS DOV:RTS

>140 .DEQ LDA IAC1 + 1:SBC IAC2 + 1:BVS DOV:RTS

>150 .DOV EOR #&80:ORA #1:RTS

>160 .GREATER JSR COMPARE:BMI TRU:BPL FALS

>170 .LESS JSR COMPARE:BMI FALS:BEQ FALS:BNE TRU

>180 .LESS__EQUAL JSR COMPARE:BPL FALS:BMI TRU

>190 .GREATER__EQUAL JSR COMPARE:BEQ TRU:BMI TRU:BPL FALS

>200 .MULTIPLY

>210 LDA #0:STA &31:STA &32

>220 LDX #16:JJ ASL &31:ROL &32:ASL IAC2:ROL IAC2 + 1:BCC KK:LDA IAC1:CLC:ADC &31:STA &31

>230 LDA IAC1 + 1:ADC &32:STA &32:..KK DEX:BNE JJ:LDA &31:STA IAC1:LDA &32:STA IAC1 + 1:RTS

>240 .PNTNUM LDA IAC1 + 1:BPL EE:LDA #ASC"-":JSR &FFEE:LDA #0:SEC:SBC IAC1:STA IAC1:LDA #0:SBC IAC1 + 1:STA IAC1 + 1

>250 .EE LDY #0:..FF LDX #16:LDA #0:..GG ASL IAC1:ROL IAC1 + 1:ROL A:CMP #10:BCC HH:SBC #10:INC IAC1:..HH DEX

>260 BNE GG:PHA:INY:LDA IAC1 + 1:ORA IAC1:BNE FF:..II PLA:CLC:ADC #&30:JSR &FFEE:DEY:BNE II:RTS

>270 .PNTSTNG PLA:STA IAC1:PLA:STA IAC1 + 1:LDY #0:..ZA INC


```

IAC1:BNE ZB:INC IAC1 + 1:ZB LDA (IAC1),Y:BEQ ZC:JSR &FFEE:
JMP ZA:ZC LDA IAC1 + 1:PHA:LDA IAC1:PHA:RTS

```

```
> 280 /End of run time library
```

```
> 290 .START%
```

```
> 300 /A demonstration of looping
```

```
> 310 ]
```

```
> 320 FRED = &54
```

```
> 330 [OPT T%
```

```
> 340 LDA #0 MOD 256:STA FRED:LDA #0 DIV 256:STA FRED + 1
```

```
> 350 .L000
```

```
> 360 INC FRED:BNE L001:INC FRED + 1:.L001
```

```
> 370 LDA FRED:STA IAC2:LDA FRED + 1:STA IAC2 + 1
```

```
> 380 LDA #32767 MOD 256:STA IAC1:LDA #32767 DIV 256:STA
IAC1 + 1
```

```
> 390 JSR EQUAL
```

```
> 400 LDA IAC1:ORA IAC1 + 1
```

```
> 410 ]:IF (P%-L000)< = 126 THEN [OPT T%:BEQ L000:] ELSE [OPT
T%:BNE L003:JMP L000:.L003:]
```

```
> 420 [OPT T%
```

```
> 430 RTS
```

```
> 440 ]NEXT T%
```

```
> 450 CALL START%
```

**** The program is now run. This does the compilation**

```
> RUN
```

**** Timings can now be taken ignoring compilation time by calling the machine code directly**

```
> TIME = 0:CALLSTART%:PRINTTIME
```

```
117
```

**** I then timed the BASIC equivalent. The results speak for themselves**

```
> TIME = 0:FRED% = 0:REPEAT FRED% = FRED% + 1:UNTIL
FRED% = 32767:PRINT TIME
```

```
7995
```

```
> PRINT 7995/117
```

```
68.3333333
```

```
>
```

As you can see, in this case, SLUG is about 70 times faster than BASIC.

With the above rigmarole to go through before the program can actually be run, it is clear that SLUG is not a language for spontaneous programming. You can always verify algorithms in BASIC, then convert them to SLUG for faster running time.

Assembly language programmers will notice that the code generated by the compiler is not amazingly efficient. In comparison to BASIC, of course, this doesn't matter.

I spent a few minutes hand optimizing the above code, and came up with this:

> LIST

```

10 REM Filename:OPTLOOP
20
30 REM Optimized loop program
40
50 REM (c) 1983 Jeremy Ruston
60
70 HIMEM = TOP + 500
80 IAC1 = &50
90 IAC2 = &52
100 FOR T% = 0 TO 2 STEP 2
110 P% = HIMEM
120 [OPT T%
130 /Run time library
140 /End of run time library
150 .START%
160 /A demonstration of looping
170 ]
180 FRED = &54
190 [OPT T%
200 LDA #0 MOD 256:STA FRED:LDA #0 DIV 256:STA FRED + 1
210 .L000
220 INC FRED:BNE L001:INC FRED + 1:.L001
230 LDA FRED:CMP #32767 MOD 256:BNE L000
240 LDA FRED + 1:CMP #32767 DIV 256:BNE L000
250 RTS
260 ]NEXT T%
270 CALL START%

```

> RUN

```

> TIME = 0:CALL START%:PRINT TIME
      29

```

>

Thus, code generated by SLUG is about 4 times slower than assembly language.

> LIST

```

10 REM Filename:SLUG
20
30 REM Structured Language with Universal Greatness
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 7
80
90 ON ERROR CLOSE#0:REPORT:PRINT " at line ";ERL:END
100
110 PROCquestions

```

```

120
130 PROCinit
140
150 PROCstatement(FNget)
160
170 PROCterm
180
190 CLEAR
200 END
210
220 DEF PROCquestions
230 LOCAL FILE$
240 PRINT ""Jeremy Ruston's""
250 PRINT "Structured"
260 PRINT "Language with"
270 PRINT "Universal"
280 PRINT "Greatness""
290 INPUT "Enter source filename:" FILE$
300 IN% = OPENIN(FILE$)
310 IF IN% = 0 THEN PRINT "Cannot open file":END
320 INPUT "Enter object filename:" FILE$
330 OUT% = OPENOUT(FILE$)
340 IF OUT% = 0 THEN PRINT "Cannot open file":END
350 ENDPROC
360
370 DEF PROCinit
380 DIM VAR$(20),PRC$(50)
390 VPT% = 0
400 PPT% = 0
410 LAB% = 0
420 LIN% = 10
430 CUR% = &54
440 EXCESS$ = ""
450 PROClibrary
460 ENDPROC
470
480 DEF FNget
490 LOCAL A$
500 IF EXCESS$ < > "" THEN A$ = EXCESS$:EXCESS$ = "": = A$
510 A$ = ""
520 REPEAT
530 A$ = A$ + CHR$(BGET #IN%)
540 UNTIL RIGHT$(A$,1) = CHR$(13)
550 A$ = LEFT$(A$,LEN(A$)-1)
560 IF LEFT$(A$,1) = " " THEN A$ = MID$(A$,2):GOTO 560

```



```

570 IF A$ = "" THEN GOTO 510
580 IF LEFT$(A$,1) = ":" THEN PROCput("/" + MID$(A$,2)):
GOTO 510
590 = A$
600
610 DEF PROCput(A$)
620 PROCplace(STR$(LIN%) + " " + A$ + CHR$(13))
630 LIN% = LIN% + 10
640 ENDPROC
650
660 DEF PROCplace(A$)
670 LOCAL T%
680 FOR T% = 1 TO LEN(A$)
690 BPUT #OUT%,ASC(MID$(A$,T%))
700 VDU ASC(MID$(A$,T%))
710 NEXT T%
720 VDU 10
730 ENDPROC
740
750 DEF FNlabel
760 LOCAL L$
770 L$ = STR$(LAB%)
780 IF LEN(L$) < 3 THEN L$ = "0" + L$:GOTO 780
790 LAB% = LAB% + 1
800 = "L" + L$
810
820 DEF PROClibrary
830 DATA HIMEM = TOP + 500
840 DATA IAC1 = &50
850 DATA IAC2 = &52
860 DATA FOR T% = 0 TO 2 STEP 2
870 DATA P% = HIMEM
880 DATA [OPT T%
890 DATA /Run time library
900 DATA .FALS LDA #0:STA IAC1:STA IAC1 + 1:RTS
910 DATA .TRU LDA #&FF:STA IAC1:STA IAC1 + 1:RTS
920 DATA .EQUAL LDA IAC1:CMP IAC2:BNE FALS:LDA
IAC1 + 1:CMP IAC2 + 1:BNE FALS:BEQ TRU
930 DATA .NO_EQUAL LDA IAC1:CMP IAC2:BNE TRU:LDA
IAC1 + 1:CMP IAC2 + 1:BNE TRU:BEQ FALS
940 DATA .COMPARE LDA IAC2:CMP IAC1:BEQ DEQ
950 DATA LDA IAC2 + 1:SBC IAC1 + 1:ORA #1:BVS DOV:RTS
960 DATA .DEQ LDA IAC2 + 1:SBC IAC1 + 1:BVS DOV:RTS
970 DATA .DOV EOR #&80:ORA #1:RTS
980 DATA .GREATER JSR COMPARE:BMI TRU:BPL FALS

```

```

990 DATA .LESS JSR COMPARE:BMI FALS:BEQ FALS:BNE TRU
1000 DATA .LESS_EQUAL JSR COMPARE:BPL FALS:BMI TRU
1010 DATA .GREATER_EQUAL JSR COMPARE:BEQ TRU:BMI
TRU:BPL FALS
1020 DATA .MULTIPLY
1030 DATA LDA #0:STA &31:STA &32
1040 DATA LDX #16:JJ ASL &31:ROL &32:ASL IAC2:ROL IAC
2 + 1:BCC KK:LDA IAC1:CLC:ADC &31:STA &31
1050 DATA LDA IAC1 + 1:ADC &32:STA &32:..KK DEX:BNE JJ:LDA
&31:STA IAC1:LDA &32:STA IAC1 + 1:RTS
1060 DATA .PNTNUM LDA IAC1 + 1:BPL EE:LDA #ASC"-":JSR
&FFEE:LDA #0:SEC:SBC IAC1:STA IAC1:LDA #0:SBC IAC1 + 1:
STA IAC1 + 1
1070 DATA .EE LDY #0:..FF LDX #16:LDA #0:..GG ASL IAC1:
ROL IAC1 + 1:ROL A:CMP #10:BCC HH:SBC #10:INC IAC1:..HH
DEX
1080 DATA BNE GG:PHA:INY:LDA IAC1 + 1:ORA IAC1:BNE
FF:..II PLA:CLC:ADC #&30:JSR &FFEE:DEY:BNE II:RTS
1090 DATA ".PNTSTNG PLA:STA IAC1:PLA:STA IAC1 + 1:LDY #0:
.ZA INC IAC1:BNE ZB:INC IAC1 + 1:..ZB LDA (IAC1),Y:BEQ ZC:JSR
&FFEE:JMP ZA:..ZC LDA IAC1 + 1:PHA:LDA IAC1:PHA:RTS"
1100 DATA /End of run time library
1110 DATA .START%
1120 DATA ***
1130 LOCAL A$
1140 REPEAT
1150 READ A$
1160 IF A$ < > "****" THEN PROCput(A$)
1170 UNTIL A$ = "****"
1180 ENDPROC
1190
1200 DEF PROCterm
1210 PROCput("RTS")
1220 PROCput("]NEXT T%")
1230 PROCput("CALL START%")
1240 CLOSE#0
1250 ENDPROC
1260
1270 DEF FNcur = MID$(L$,PNT%,1)
1280
1290 DEF FNnext
1300 PNT% = PNT% + 1
1310 = MID$(L$,PNT%-1,1)
1320
1330 DEF PROCspaces

```

```

1340 LOCAL D$
1350 IF FNcur = " " THEN D$ = FNnext:GOTO 1350
1360 ENDPROC
1370
1380 DEF FNnumber
1390 LOCAL A$
1400 PROCspaces
1410 A$ = ""
1420 IF FNcur > "9" OR FNcur < "0" THEN = A$
1430 A$ = A$ + FNnext
1440 GOTO 1420
1450
1460 DEF FNvar__name
1470 LOCAL A$
1480 PROCspaces
1490 A$ = ""
1500 IF FNcur < "A" OR (FNcur > "Z" AND FNcur < "__") OR FNcur
> "z" THEN = A$
1510 A$ = A$ + FNnext
1520 IF FNcur < "0" OR (FNcur > "9" AND FNcur < "A") OR (FNcur-
> "Z" AND FNcur < "__") OR FNcur > "z" THEN = A$
1530 GOTO 1510
1540
1550 DEF PROCcheck(A$)
1560 LOCAL T%,F%
1570 IF A$ = "" THEN PROCbad__var
1580 IF VPT% = 0 THEN PROCno__var
1590 F% = -1
1600 FOR T% = 0 TO VPT%-1
1610 IF A$ = VAR$(T%) THEN F% = T%
1620 NEXT T%
1630 IF F% = -1 THEN PROCno__var
1640 ENDPROC
1650
1660 DEF PROCerror(A$)
1670 PRINT "Error - ";A$;" in "'L$
1680 CLOSE#0
1690 END
1700
1710 DEF PROCno__var PROCerror("No such variable")
1720 DEF PROCbad__ter PROCerror("Bad terminator")
1730 DEF PROCbad__var PROCerror("Bad variable")
1740
1750 DEF FNsearch(A$)
1760 IF MID$(L$,PNT%,LEN(A$)) = A$ THEN PNT% = PNT% +

```



```

LEN(A$): = TRUE ELSE = FALSE
1770
1780 DEF PROCstatement(L$)
1790 PNT% = 1
1800 IF FNsearch("BEGIN") THEN PROCbegin:ENDPROC
1810 IF FNsearch("IF") THEN PROCif:ENDPROC
1820 IF FNsearch("!") THEN PROCword__poke:ENDPROC
1830 IF FNsearch("?") THEN PROCbyte__poke:ENDPROC
1840 IF FNsearch("WHILE") THEN PROCwhile:ENDPROC
1850 IF FNsearch("UNTIL") THEN PROCuntil:ENDPROC
1860 IF FNsearch("EXIT") THEN PROCexit:ENDPROC
1870 IF FNsearch("VAR") THEN PROCvar:ENDPROC
1880 IF FNsearch("OUTPUT") THEN PROCoutput:ENDPROC
1890 IF FNsearch("INC") THEN PROCinc:ENDPROC
1900 IF FNsearch("DEC") THEN PROCdec:ENDPROC
1910 IF FNsearch("ASM") THEN PROCasm:ENDPROC
1920 IF FNsearch("WRITE") THEN PROCwrite:ENDPROC
1930 PROCassign
1940 ENDPROC
1950
1960 DEF PROCbegin
1970 PROCspaces
1980 IF FNcur<>" " THEN PROCbad__ter
1990 L$ = FNget
2000 IF L$ = "END" THEN ENDPROC
2010 PROCstatement(L$)
2020 GOTO 1990
2030
2040 DEF PROCexit
2050 PROCspaces
2060 IF FNcur<>" " THEN PROCbad__ter
2070 PROCput("RTS")
2080 ENDPROC
2090
2100 DEF PROCvar
2110 LOCAL A$
2120 PROCput("]")
2130 A$ = FNvar__name
2140 IF A$ = "" THEN PROCbad__var
2150 VAR$(VPT%) = A$
2160 VPT% = VPT% + 1
2170 PROCput(A$ + " = &" + STR$(CUR%))
2180 CUR% = CUR% + 2
2190 PROCspaces
2200 IF FNcur = "" THEN PROCput("[OPT T%"):ENDPROC

```

```

2210 IF FNnext<>" ," THEN PROCerror("Missing ,")
2220 GOTO 2130
2230
2240 DEF PROCinc
2250 LOCAL A$,LAB$
2260 A$ = FNvar__name
2270 PROCcheck(A$)
2280 LAB$ = FNlabel
2290 PROCput("INC " + A$ + ":BNE " + LAB$ + ":INC " + A$ + " + 1:."
+ LAB$)
2300 ENDPROC
2310
2320 DEF PROCdec
2330 LOCAL A$,LAB$
2340 A$ = FNvar__name
2350 PROCcheck(A$)
2360 LAB$ = FNlabel
2370 PROCput("DEC " + A$ + ":BNE " + LAB$ + ":DEC " + A$ + " + 1:."
+ LAB$)
2380 ENDPROC
2390
2400 DEF PROCasm
2410 PROCspaces
2420 IF FNnext<>"" THEN PROCbad__ter
2430 L$ = FNget
2440 IF L$ = "ENDASM" THEN ENDPROC
2450 PROCput(L$)
2460 GOTO 2430
2470
2480 DEF PROCassign
2490 LOCAL A$
2500 A$ = FNvar__name
2510 PROCcheck(A$)
2520 PROCspaces
2530 IF FNnext<>" =" THEN PROCerror("Missing =")
2540 PROCexp
2550 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
STA " + A$ + ":LDA #" + RLT$ + " DIV 256:STA " + A$ + " + 1"):ENDP
ROC
2560 PROCput("LDA " + RLT$ + ":STA " + A$ + ":LDA " + RLT
$ + " + 1:STA " + A$ + " + 1"):ENDPROC
2570
2580 DEF PROCoutput
2590 LOCAL A$
2600 PROCexp

```

```

2610 PROCspaces
2620 A$ = FNnext
2630 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
JSR &FFEE")
2640 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + ":JSR &FFEE")
2650 IF A$ < > ";" THEN GOTO 2700
2660 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " DIV 256:
JSR &FFEE")
2670 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + " + 1:JSR &
FFEE")
2680 PROCspaces
2690 IF FNcur = "" THEN ENDPROC ELSE GOTO 2600
2700 IF A$ = "," THEN GOTO 2600
2710 PROCspaces
2720 IF FNcur < > "" THEN PROCbad__ter
2730 ENDPROC
2740
2750 DEF PROCuntil
2760 LOCAL DEXP$,START$,FIN$
2770 DEXP$ = MID$(L$,PNT%)
2780 START$ = FNlabel
2790 PROCput("." + START$)
2800 PROCstatement(FNget)
2810 L$ = DEXP$
2820 PNT% = 1
2830 PROCexp
2840 IF FNcur < > "" THEN PROCbad__ter
2850 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
ORA #" + RLT$ + " DIV 256")
2860 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + ":ORA " + RLT$
+ " + 1")
2870 FIN$ = FNlabel
2880 PROCput("]:IF (P%- " + START$ + ") < = 126 THEN [OPT T%:
BEQ " + START$ + ":] ELSE [OPT T%:BNE " + FIN$ + ":JMP " +
START$ + "::. " + FIN$ + ":]")
2890 PROCput("[OPT T%")
2900 ENDPROC
2910
2920 DEF PROCwhile
2930 LOCAL BEG$,FIN$,START$
2940 BEG$ = FNlabel
2950 PROCput("." + BEG$)
2960 PROCexp
2970 IF FNcur < > "" THEN PROCbad__ter
2980 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:

```



```

ORA # " + RLT$ + " DIV 256")
2990 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + ":ORA " + RLT$
+ " + 1")
3000 START$ = FNlabel
3010 FIN$ = FNlabel
3020 PROCput("BNE " + START$ + ":JMP " + FIN$ + ":@" + START$)
3030 PROCstatement(FNget)
3040 PROCput("JMP " + BEG$ + ":@" + FIN$)
3050 ENDPROC
3060
3070 DEF PROCif
3080 LOCAL ELS$,NEQ$
3090 PROCexp
3100 PROCspaces
3110 IF MID$(L$,PNT%,4)<>"THEN" THEN PROCerror("Miss-
ing THEN")
3120 PNT% = PNT% + 4
3130 PROCspaces
3140 IF FNcur<>" " THEN PROCbad__ter
3150 IF RLT% = 0 THEN PROCput("LDA # " + RLT$ + " MOD 256:
ORA # " + RLT$ + " DIV 256")
3160 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + ":ORA " + RLT$
+ " + 1")
3170 NEQ$ = FNlabel
3180 PROCput("BNE P% + 5:JMP " + NEQ$)
3190 PROCstatement(FNget)
3200 L$ = FNget
3210 PNT% = 1
3220 IF LEFT$(L$,4) = "ELSE" THEN GOTO 3260
3230 EXCESS$ = L$
3240 PROCput("." + NEQ$)
3250 ENDPROC
3260 PNT% = PNT% + 4
3270 PROCspaces
3280 IF FNcur<>" " THEN PROCbad__ter
3290 ELS$ = FNlabel
3300 PROCput("JMP " + ELS$ + ":@" + NEQ$)
3310 PROCstatement(FNget)
3320 PROCput("." + ELS$)
3330 ENDPROC
3340
3350 DEF PROCexp
3360 PROClevel__1
3370 IF FNsearch("OR") THEN PROCor:ENDPROC
3380 IF FNsearch("EOR") THEN PROCeor:ENDPROC

```

```

3390 ENDPROC
3400
3410 DEF PROClevel__1
3420 PROClevel__2
3430 IF FNsearch("AND") THEN PROCand:ENDPROC
3440 ENDPROC
3450
3460 DEF PROClevel__2
3470 PROClevel__3
3480 IF FNsearch("=") THEN PROCequal:ENDPROC
3490 IF FNsearch("< >") THEN PROCnot__equal:ENDPROC
3500 IF FNsearch("> =") THEN PROCgreater__equal:ENDPROC
3510 IF FNsearch("< =") THEN PROCless__equal:ENDPROC
3520 IF FNsearch("> ") THEN PROCgreater:ENDPROC
3530 IF FNsearch("< ") THEN PROCless:ENDPROC
3540 ENDPROC
3550
3560 DEF PROClevel__3
3570 PROClevel__4
3580 IF FNsearch("+ ") THEN PROCadd:ENDPROC
3590 IF FNsearch("- ") THEN PROCsub:ENDPROC
3600 ENDPROC
3610
3620 DEF PROClevel__4
3630 PROCoperand
3640 IF FNsearch("*") THEN PROCtimes:ENDPROC
3650 IF FNsearch("DIV") THEN PROCdiv:ENDPROC
3660 IF FNsearch("MOD") THEN PROCmod:ENDPROC
3670 ENDPROC
3680
3690 DEF PROCoperand
3700 PROCspaces
3710 IF FNsearch("- ") THEN PROCunary__minus:ENDPROC
3720 IF FNsearch("+ ") THEN PROCoperand:ENDPROC
3730 IF FNsearch("(") THEN PROCbracket:ENDPROC
3740 IF FNsearch("?") THEN PROCbyte__ind:ENDPROC
3750 IF FNsearch("!") THEN PROCword__ind:ENDPROC
3760 IF FNsearch("GET") THEN PROCput("LDA #124:JSR &FFF4:
JSR &FFE0:STA IAC1:LDA #0:STA IAC1 + 1"):RLT% = 1:RLT$ = "
IAC1":ENDPROC
3770 RLT$ = FNnumber
3780 IF RLT$<>" " THEN RLT% = 0:PROCspaces:ENDPROC
3790 RLT$ = FNvar__name
3800 PROCcheck(RLT$)
3810 RLT% = 1

```

```

3820 PROCspaces
3830 ENDPROC
3840
3850 DEF PROCpush__iac
3860 PROCput("LDA IAC1:PHA:LDA IAC1 + 1:PHA")
3870 ENDPROC
3880
3890 DEF PROCpull__iac
3900 PROCput("PLA:STA IAC2 + 1:PLA:STA IAC2")
3910 ENDPROC
3920
3930 DEF PROCsub
3940 PROCmaths("SBC","SEC:SBC")
3950 ENDPROC
3960
3970 DEF PROCadd
3980 PROCmaths("ADC","CLC:ADC")
3990 ENDPROC
4000
4010 DEF PROCand
4020 PROCmaths("AND","AND")
4030 ENDPROC
4040
4050 DEF PROCeor
4060 PROCmaths("EOR","EOR")
4070 ENDPROC
4080
4090 DEF PROCor
4100 PROCmaths("ORA","ORA")
4110 ENDPROC
4120
4130 DEF PROCmaths(X$,Y$)
4140 LOCAL OLT%,OLT$
4150 IF RLT% = 1 AND RLT$ = "IAC1" THEN PROCpush__iac
4160 OLT% = RLT%:OLT$ = RLT$
4170 IF X$ = "ADC" OR X$ = "SBC" THEN PROClevel__3
4180 IF X$ = "ORA" OR X$ = "EOR" THEN PROCexp
4190 IF X$ = "AND" THEN PROClevel__1
4200 IF OLT% = 1 AND OLT$ = "IAC1" THEN PROCpull__iac:
OLT$ = "IAC2"
4210 IF OLT% = 0 AND RLT% = 0 AND X$ = "ORA" THEN RLT$ =
STR$(VAL(OLT$) OR VAL(RLT$)):ENDPROC
4220 IF OLT% = 0 AND RLT% = 0 AND X$ = "EOR" THEN RLT$ =
STR$(VAL(OLT$) EOR VAL(RLT$)):ENDPROC
4230 IF OLT% = 0 AND RLT% = 0 AND X$ = "AND" THEN RLT$ =

```



```

STR$(VAL(OLT$) AND VAL(RLT$)):ENDPROC
4240 IF OLT% = 0 AND RLT% = 0 AND X$ = "ADC" THEN RLT$ =
STR$(VAL(OLT$) + VAL(RLT$)):ENDPROC
4250 IF OLT% = 0 AND RLT% = 0 AND X$ = "SBC" THEN RLT$ =
STR$(((VAL(OLT$)-VAL(RLT$)) + 65536) MOD 65536):ENDPROC
4260 IF OLT% = 0 AND RLT% = 1 THEN PROCput("LDA #" + OLT$
+ " MOD 256:" + Y$ + " " + RLT$ + ":STA IAC1:LDA #" + OLT$ + "
DIV 256:" + X$ + " " + RLT$ + " + 1:STA IAC1 + 1"):RLT$ = "IAC1":END
PROC
4270 IF OLT% = 1 AND RLT% = 0 THEN PROCput("LDA " + OLT$
+ ":" + Y$ + " #" + RLT$ + " MOD 256:STA IAC1:LDA " + OLT$ + " + 1:
" + X$ + " #" + RLT$ + " DIV 256:STA IAC1 + 1"):RLT$ = "IAC1":RLT%
= 1:ENDPROC
4280 IF OLT% = 1 AND RLT% = 1 THEN PROCput("LDA " + RLT$
+ ":" + Y$ + " " + OLT$ + ":STA IAC1:LDA " + RLT$ + " + 1:" + X$ + " " +
OLT$ + " + 1:STA IAC1 + 1"):RLT$ = "IAC1":ENDPROC
4290
4300 DEF PROCequal
4310 PROCcomparison("EQUAL")
4320 ENDPROC
4330
4340 DEF PROCnot_equal
4350 PROCcomparison("NO_EQUAL")
4360 ENDPROC
4370
4380 DEF PROCgreater
4390 PROCcomparison("GREATER")
4400 ENDPROC
4410
4420 DEF PROCless
4430 PROCcomparison("LESS")
4440 ENDPROC
4450
4460 DEF PROCgreater_equal
4470 PROCcomparison("GREATER_EQUAL")
4480 ENDPROC
4490
4500 DEF PROCless_equal
4510 PROCcomparison("LESS_EQUAL")
4520 ENDPROC
4530
4540 DEF PROCcomparison(OP$)
4550 LOCAL OLT%,OLT$,BRN$
4560 IF RLT% = 1 AND RLT$ = "IAC1" THEN PROCpush_iac
4570 OLT% = RLT%:OLT$ = RLT$

```

```

4580 PROClevel__2
4590 PROCdo__it(OP$)
4600 ENDPROC
4610
4620 DEF PROCtimes
4630 LOCAL OLT%,OLT$,BRN$
4640 IF RLT% = 1 AND RLT$ = "IAC1" THEN PROCpush__iac
4650 OLT% = RLT%:OLT$ = RLT$
4660 PROClevel__4
4670 PROCdo__it("MULTIPLY")
4680 ENDPROC
4690
4700 DEF PROCdo__it(OP$)
4710 IF OLT% = 1 AND OLT$ = "IAC1" THEN PROCpull__iac:OLT$
= "IAC2"
4720 BRN$ = FNlabel
4730 IF OLT% = 0 AND RLT% = 0 THEN RLT$ = STR$((65536 +
(RLT$ = OLT$)) MOD 65536):ENDPROC
4740 IF OLT% = 0 THEN PROCput("LDA #" + OLT$ + " MOD 256:
STA IAC2:LDA #" + OLT$ + " DIV 256:STA IAC2 + 1"):OLT% = 1:
OLT$ = "IAC2"
4750 IF OLT% = 1 AND OLT$ <> "IAC2" THEN PROC put("LDA " +
OLT$ + ":STA IAC2:LDA " + OLT$ + " + 1:STA IAC2 + 1"):OLT$ = "IAC2"
4760 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
STA IAC1:LDA #" + RLT$ + " DIV 256:STA IAC1 + 1"):RLT% = 1:RLT$
= "IAC1"
4770 IF RLT% = 1 AND RLT$ <> "IAC1" THEN PROCput("LDA
" + RLT$ + ":STA IAC1:LDA " + RLT$ + " + 1:STA IAC1 + 1"):RLT$ =
"IAC1"
4780 PROCput("JSR " + OP$)
4790 RLT% = 1:RLT$ = "IAC1"
4800 ENDPROC
4810
4820 DEF PROCunary__minus
4830 PROCoperand
4840 IF RLT% = 0 THEN RLT$ = STR$(65536-VAL(RLT$)):ENDPROC
4850 PROCput("LDA #0:SEC:SBC " + RLT$ + ":STA IAC1:LDA #0:
SBC " + RLT$ + " + 1:STA IAC1 + 1")
4860 RLT$ = "IAC1"
4870 ENDPROC
4880
4890 DEF PROCbracket
4900 PROCexp
4910 IF FNnext<>" )" THEN PROCerror("Missing ")

```

```

4920 ENDPROC
4930
4940 DEF PROCbyte__ind
4950 PROCoperand
4960 IF RLT% = 0 THEN PROCput("LDA " + RLT$ + ":STA IAC1:
LDA #0:STA IAC1 + 1"):RLT% = 1:RLT$ = "IAC1":ENDPROC
4970 PROCput("LDY #0:LDA (" + RLT$ + "),Y:STA IAC1:LDA #0:
STA IAC1 + 1"):RLT$ = "IAC1":ENDPROC
4980
4990 DEF PROCword__ind
5000 PROCoperand
5010 IF RLT% = 0 THEN PROCput("LDA " + RLT$ + ":STA IAC1:
LDA " + RLT$ + " + 1:STA IAC1 + 1"):RLT% = 1:RLT$ = "IAC1":END
PROC
5020 PROCput("LDY #0:LDA (" + RLT$ + "),Y:STA IAC1:INY:LDA ("
+ RLT$ + "),Y:STA IAC1 + 1"):RLT$ = "IAC1":ENDPROC
5030
5040 DEF PROCwrite
5050 PROCspaces
5060 IF FNcur = "" THEN ENDPROC
5070 IF FNcur = CHR$(34) THEN GOTO 5380
5080 IF FNsearch("TAB") THEN GOTO 5230
5090 IF FNcur = "," THEN GOTO 5200
5100 IF FNcur = "" THEN GOTO 5170
5110 IF FNcur = ";" THEN PNT% = PNT% + 1:GOTO 5050
5120 PROCspaces
5130 PROCexp
5140 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
STA IAC1:LDA #" + RLT$ + " DIV 256:STA IAC1 + 1:JSR PNTNUM")
5150 IF RLT% = 1 THEN PROCput("LDA " + RLT$ + ":STA IAC1:
LDA " + RLT$ + " + 1:STA IAC1 + 1:JSR PNTNUM")
5160 GOTO 5050
5170 PROCput("JSR &FFE7")
5180 PNT% = PNT% + 1
5190 GOTO 5050
5200 PNT% = PNT% + 1
5210 PROCput("LDA #32:JSR &FFEE")
5220 GOTO 5050
5230 IF FNnext<>"(" THEN PROCerror("Missing ( after TAB")
5240 PROCexp
5250 PROCspaces
5260 IF FNcur = "," THEN GOTO 5310
5270 IF FNnext<>")" THEN PROCerror("Missing ) after TAB")
5280 IF RLT% = 1 THEN PROCput("LDA #134:JSR &FFF4:LDA #31
:JSR &FFEE:LDA " + RLT$ + ":JSR &FFEE:TYA:JSR &FFEE")

```



```

5290 IF RLT% = 0 THEN PROCput("LDA #134:JSR &FFF4:LDA #31
:JSR &FFEE:LDA #" + RLT$ + " MOD 256:JSR &FFEE:TYA:JSR &
FFEE")
5300 GOTO 5050
5310 PROCput("LDA #31:JSR &FFEE:LDA IAC1:JSR &FFEE")
5320 PNT% = PNT% + 1
5330 PROCexp
5340 PROCspaces
5350 IF FNnext<>"") THEN PROCerror("Missing ) after TAB")
5360 IF RLT% = 0 THEN PROCput("LDA #" + RLT$ + " MOD 256:
JSR &FFEE") ELSE PROCput("LDA " + RLT$ + ":JSR &FFEE")
5370 GOTO 5050
5380 PROCput("JSR PNTSTNG")
5390 LOCAL D$
5400 D$ = ""
5410 IF FNcur = "" THEN PROCerror("Missing " + CHR$(34))
5420 D$ = D$ + FNnext
5430 IF RIGHT$(D$,1)<>CHR$(34) OR D$ = CHR$(34) THEN GOTO
5410
5440 PROCput("]$P% = " + D$ + " + CHR$(0):P% = P% + " + STR$(LEN
(D$)-1) + ":[OPT T%)")
5450 GOTO 5050
5460
5470 DEF PROCword__poke
5480 LOCAL OLT%,OLT$
5490 PROCoperand
5500 PROCspaces
5510 IF FNnext<>" = " THEN PROCerror("Missing = ")
5520 IF RLT% = 1 AND RLT$ = "IAC1" THEN PROCpush__iac
5530 OLT% = RLT%:OLT$ = RLT$
5540 PROCexp
5550 PROCspaces
5560 IF OLT% = 1 AND OLT$ = "IAC1" THEN PROCpull__iac:OLT$
= "IAC2"
5570 IF FNcur<>"" THEN PROCbad__ter
5580 IF RLT% = 0 AND OLT% = 0 THEN PROCput("LDA #" + RLT$
+ " MOD 256:STA " + OLT$ + ":LDA #" + RLT$ + " DIV 256:STA " +
OLT$ + " + 1")
5590 IF RLT% = 1 AND OLT% = 0 THEN PROCput("LDA " + RLT$
+ ":STA " + OLT$ + ":LDA " + RLT$ + " + 1:STA " + OLT$ + " + 1")
5600 IF RLT% = 0 AND OLT% = 1 THEN PROCput("LDY #0:LDA
#" + RLT$ + " MOD 256:STA (" + OLT$ + "),Y:INY:LDA #" + RLT$ + "
DIV 256:STA (" + OLT$ + "),Y")
5610 IF RLT% = 1 AND OLT% = 1 THEN PROCput("LDY #0:LDA "
+ RLT$ + ":STA (" + OLT$ + "),Y:INY:LDA " + RLT$ + " + 1:STA (" +

```

```

OLT$
+ "),Y")
5620 ENDPROC
5630
5640 DEF PROCbyte__poke
5650 LOCAL OLT%,OLT$
5660 PROCoperand
5670 PROCspaces
5680 IF FNnext<>"=" THEN PROCerror("Missing =")
5690 IF RLT%=1 AND RLT$="IAC1" THEN PROCpush__iac
5700 OLT%=RLT%:OLT$=RLT$
5710 PROCexp
5720 PROCspaces
5730 IF OLT%=1 AND OLT$="IAC1" THEN PROCpull__iac:OLT$
= "IAC2"
5740 IF FNcur<>"" THEN PROCbad__ter
5750 IF RLT%=0 AND OLT%=0 THEN PROCput("LDA #" + RLT$
+ " MOD 256:STA " + OLT$)
5760 IF RLT%=1 AND OLT%=0 THEN PROCput("LDA " + RLT$
+ ":STA " + OLT$)
5770 IF RLT%=0 AND OLT%=1 THEN PROCput("LDY #0:LDA
#" + RLT$ + " MOD 256:STA (" + OLT$ + "),Y")
5780 IF RLT%=1 AND OLT%=1 THEN PROCput("LDY #0:LDA "
+ RLT$ + ":STA (" + OLT$ + "),Y")
5790 ENDPROC
>

```

Line by line notes:

Line 90 sets up a simple error handler. It prints the error message as normal, but also closes all files.

Line 110 calls PROCquestions which gets the source and object filenames from the user.

Line 130 calls PROCinit which initialises some of the global variables used by the compiler.

Line 150 calls PROCstatement. PROCstatement will compile a single statement - even if it is a compound statement. The first line of the statement is passed as a parameter. The statement is got from FNget, which returns the next source line.

Line 170 calls PROCterm which adds the closing dialogue to the object program.

Line 190 clears all variables before ending. This is done to allow you to move straight into MODE 3 when the compiler ends, for easy editing.

Line 200 terminates the program.

Line 220 starts the definition of PROCquestions.

Lines 240 to 280 print the sign on message.

Line 290 asks for the source filename.

Line 300 tries to open the source file.

Line 310 terminates the program if the source file could not be opened.

Line 320 gets the name of the output file.

Line 330 tries to open the file.

Line 340 checks to see if the file could be opened.

Line 350 exits PROCquestions.

Line 370 starts the definition of PROCinit.

Line 380 dimensions the two string arrays used by the program. The first, VAR\$, is used to hold the names of all the variables used in the program. The second, PRC\$, is not used in this version of SLUG but normally holds the names of all the procedures used in the program.

Line 390 resets the pointer into VAR\$. VPT% always points to the first empty element of the array.

Line 400 resets a similar pointer for PRC\$. Again, this is not used in this version.

Line 410 defines the current label. The compiler often has to generate labels, without repeating itself. The variable LAB% is used for this purpose - see line 750.

Line 420 defines the variable LIN%, which holds the line number of the next object line to be written.

Line 430 defines the lowest unused memory location for variables. Thus, all variables are situated on page zero, for speed.

Line 440 resets the variables EXCESS\$. The function of this variable will become clear when we look at the way I have compiled IF-THEN-ELSE.

Line 450 calls PROClibrary, which defines the library routines which could be used by the program.

Line 480 starts to define FNget. This function returns the next source line.

Line 500 checks to see if EXCESS\$ is null. If it is not, it is returned as the next line.

Line 510 clears the variable A\$. The line of text will be built up in A\$.

Line 520 starts a REPEAT loop which will continue until the entire line has been read in.

Line 530 adds the next character read from the source file to A\$.

Line 540 checks to see if the end of the line has been reached by seeing if the return at the end of each line has been read in.

Line 550 slices the carriage return off the end of A\$.

Line 560 checks to see if the line begins with a space. If it does, the space is removed and the line is again inspected.

Line 570 checks to see if A\$ has become null by the process of knocking the leading spaces off. If it has, a new line is read in.

Line 580 checks to see if the current line is a comment line. If it is, the line is passed directly to the object code and a new line is sought.

Line 590 exits the function with A\$.

Line 610 defines PROCput, which puts a given string out to the object file.

Line 620 calls PROCplace, which puts any string out to the object file. The argument passed is the current line number added to the text of the line.

Line 630 increments the line number count.

THE BBC MICRO COMPENDIUM

Line 660 starts the definition of PROCplace.

Line 680 loops through all the characters of the string to be output.

Line 690 puts the character out to the file.

Line 700 prints the character.

Line 710 terminates the loop.

Line 720 prints a line feed, since the output string will have ended in a carriage return, but the line feed has to be artificially added.

Line 750 starts to define FNlabel. This function returns, as a string, the next unused label. Labels are of the form of a three digit number preceded by the letter 'L'.

Line 770 converts the current label count to a string.

Line 780 pads it out to the left with zeros.

Line 790 increments the label counter.

Line 800 returns with the string form of the label, and a letter 'L'.

Line 820 starts to define PROClibrary. The DATA statements contain the first few lines of the object program.

Line 830 lowers HIMEM to leave space for the machine code.

Lines 840 and 850 define the locations of two special variables. These are the accumulators used by the system, like the FROTH accumulators.

Line 860 loops through the OPT values necessary to disable a listing but trap errors.

Line 870 sets the code counter to HIMEM.

Lines 900 to 1090 are the library routines. Most are the same as those I used in FROTH, and so will not be commented upon individually.

lines 1130 to 1170 read in these lines, and pump them out to the object file.

Line 1200 starts to define PROCterm, which adds the post-amble to the object code.

Line 1210 places a RTS instruction.

Line 1220 makes the second pass occur.

Line 1230 inserts a line to actually run the code. All programs start with the label START%.

Line 1270 is used to return the current character of the source program. Throughout the compiler, L\$ is used to hold the current line, whilst PNT% holds the position within the line that we have reached.

Line 1290 is used to return the next character of the source program, incrementing the pointer PNT% as it does so.

Line 1330 starts to define PROCspaces. This procedure removes any spaces at the current position of the source program.

Line 1350 looks for a space. If it finds one, it copies it into a dummy character and checks again.

Line 1360 terminates the procedure.

Line 1380 defines FNnumber. This function looks for a number in the source text. If it finds one, it returns it as a string, otherwise it returns the null string.

Line 1400 removes any spaces that may precede the number.

Line 1410 sets A\$ to the null string. A\$ will be used to build up the number.

Line 1420 checks to see if the current character is a digit. If it is not, the routine exits with A\$ in tow.

Line 1430 adds the digit to the end of A\$.

Line 1440 goes back to look for more digits.

Line 1460 defines FNvar__name. This function returns a variable name extracted from the text. If one is not present, it returns the null string.

Line 1480 removes spaces from before the name.

Line 1490 sets A\$, which is used to build up the name, to the null string.

Line 1500 checks to see if the first letter encountered is a legal one with which to start a variable name. If it is not, it returns with A\$ - which is a null string at this point.

Line 1510 adds the character to A\$ if it was legal.

Line 1520 checks to see if the next character is legal for the middle of a variable name.

Line 1530 goes back to add it to A\$ if it was legal.

Line 1550 defines PROCcheck. This procedure checks if its argument is a proper variable. It also checks that the variable exists.

Line 1570 calls PROCbad__var if the variable name is null. This prints the message 'Bad variable'.

Line 1580 gives a 'No such variable' message if the variable pointer is still set to zero.

Line 1590 starts searching through all the variables so far defined to see if the current one is among them. It starts by setting a flag to -1. This flag keeps count of whether we have found the variable yet.

Line 1600 loops through all the defined variables.

Line 1610 compares the current variable from VAR\$ to the variable in question, setting the flag to the index into VAR\$ if the variable names match.

Line 1620 terminates the loop.

Line 1630 checks the flag.

Line 1660 defined PROCerror. This procedure is used to terminate the program and print an error message whenever an error is encountered.

Line 1670 prints the message and the line in which the error occurred.

Line 1680 closes all files.

Lines 1710 to 1730 are special procedures to handle errors which occur at many different points in the compiler.

Line 1750 defines FNsearch. This function checks to see if the argument string is present at the current position in the source line. If it is present, the pointer is moved passed it and TRUE is returned, else FALSE is returned.

Line 1760 does the above.

Line 1780 starts to define the most important procedure in the program, PROC-statement. This procedure takes one line of source text as a parameter, and then compiles code for the statement in the source line provided.

Line 1790 resets the pointer into the line.

Lines 1800 to 1920 check for the various statements allowed. Notice that some 'statements', like END, are not checked for. This is because they are handled by the statements that use them (eg. BEGIN).

Line 1930 assumes that the statement must be an assignment statement if nothing else matches.

Line 1960 defines the procedure for handling BEGIN.

Line 1970 removes the spaces that may be following the word BEGIN.

Line 1980 checks to see if the end of the line has been reached. If it has not, there is something following the word BEGIN that should not be there, so the 'Bad terminator' message is given.

Line 1990 gets the next source line.

Line 2000 inspects it to see if it is 'END'. If it is, the BEGIN routine is finished.

Line 2010 compiles the new source line.

Line 2020 goes back for more statements.

Line 2040 starts the definition of the procedure to handle the statement.

Line 2050 removes the spaces that may follow the word EXIT.

Line 2060 checks to make sure nothing follows the word EXIT.

Line 2070 places the RTS instruction that corresponds to EXIT in the object file.

Line 2100 defines the procedure that handles the VAR declaration statement.

Line 2120 puts a right hand square bracket into the object code. This moves out of the assembler into normal BASIC, where labels can be defined.

Line 2130 gets a variable name from the text.

Line 2140 checks to make sure that one could be found.

Line 2150 inserts the variable name into the variable table.

Line 2160 increments the pointer into VAR\$.

Line 2170 outputs code to set the current free location to the variable name. For example 'VAR H' would result in:

H = &54

Line 2180 increments the free space pointer past the newly created variable.

Line 2190 removes the spaces that may precede the comma following the variable name.

Line 2200 checks to see if the end of the statement has been reached. If it has, it outputs code to move back into the assembler.

Line 2210 ensures that a comma is present.

Line 2220 goes back for another variable name.

Line 2240 defines the procedure that deals with the INC statement.

Line 2260 gets the name of the variable to increment.

Line 2270 checks the variable.

Line 2280 gets the next available label.

Line 2290 places code in the object file to increment the variable.

Line 2300 exits the procedure.

Lines 2320 to 2380 are the same as PROCincrement, except that they decrement the locations.

Line 2400 defines the procedure to deal with the ASM statement.

Lines 2410 and 2420 check to make sure the statement is terminated correctly.

Line 2430 gets the next source line.

Line 2440 checks for the ENDASM statement. If it is found, the ASM statement exits.

Line 2450 passes the statement straight to the object file.

Line 2460 goes back for more lines.

Line 2480 starts the definition of the procedure which compiles assignment statements.

Line 2500 gets the name of the variable we are about to assign to.

Line 2510 checks it.

Line 2520 removes any spaces that may precede the equals sign.

Line 2530 ensures the equals sign is present.

Line 2540 compiles the expression following the equals sign. PROCexp will also take care of any leading spaces.

When PROCexp returns, it passes back the variables RLT% and RLT\$ which tell the calling routine where to find the answer to the expression. If RLT% is zero, the answer is a number. The number is contained, in string form, in RLT\$. If RLT% is one, RLT\$ contains the label of a location where the answer may be found.

Line 2550 deals with the case of the answer to the expression being a number. It assembles code to place the number at the indicated location.

Line 2560 does the same thing for the answer being a label.

Line 2580 starts to define the procedure which deals with the OUTPUT statement.

Line 2600 gets the expression following the word OUTPUT.

Line 2610 removes the spaces that may follow it.

Line 2620 gets the character following the expression.

Line 2630 deals with the answer to the expression being a number. It only outputs the least significant byte.

Line 2640 does the same for the result being stored at a label.

Line 2650 checks to see if the expression was followed by a semi colon. If it was, the more significant byte must also be output. If it was not, the code is skipped.

Line 2660 outputs the more significant byte of the number result.

Line 2670 outputs the more significant byte of the result stored at a label.

Line 2680 removes any spaces that may follow it.

Line 2690 checks to see if the end of the statement has been reached. If it has, the routine exits, else it goes back for another expression.

Line 2700 is where the byte statement ends up. It looks to see if the expression was followed by a comma. If it was, control is passed back to get another expression.

Line 2710 removes any spaces.

Line 2720 checks to make sure that there are no illegal characters after the expression.

Line 2730 leaves the procedure.

Line 2750 starts the procedure that deals with the UNTIL statement.

Line 2770 extracts the expression that follows the word UNTIL. It will be used after the statement following UNTIL has been compiled.

Line 2780 generates a label for the start of the statement.

Line 2790 inserts this label into the object code. Thus, if the statement has to be re-executed, a JMP can be made to this label.

Line 2800 compiles the next statement.

Line 2810 copies the expression into L\$.

Line 2820 resets the pointer to the start of the line.

Line 2830 calls PROCexp to evaluate the expression.

Line 2840 ensures that the expression is not followed by any illegal characters.

Line 2850 deals with the result of the expression being a number. The two bytes of the number are ORd together, setting the zero flag in the process. The setting of the zero flag is used to determine whether the loop should be re-executed.

Line 2860 does the same thing for a labelled result.

Line 2870 creates a label for the end of the loop.

Line 2880 inserts a rather complicated line into the BASIC program. If the zero flag is set, control is passed to the start of the program, else the program continues sequentially. The complex line is used to decide whether to use a branch or JMP instruction.

Line 2890 inserts a line to go back into assembly language mode.

Line 2920 defines the procedure that compiles the WHILE statement.

Lines 2940 to 3050 are similar to those in PROCuntil. The difference is that the expression is evaluated first. In addition, the meaning of the zero flag is reversed.

Line 3070 starts the definition of PROCif, which compiles the IF statement.

Line 3090 compiles the expression following the word IF.

Line 3100 removes any spaces that may follow the expression.

Line 3110 ensures that the word THEN appears after the expression.

Line 3120 increments the pointer past the word THEN.

Line 3130 removes any spaces that may follow the word THEN.

Line 3140 checks to make sure there is no more text after the word THEN.

Line 3150 deals with the situation where the result of an expression is a number. It sets the zero flag according to the result.

Line 3160 deals with the situation where the result is a label. The two bytes of the number are ORed together to set the zero flag according to the result.

Line 3170 creates a label to go to if the expression is FALSE.

Line 3180 inserts code to jump to NEQ\$ if the expression is FALSE, and to continue if it is TRUE.

Line 3190 compiles the statement following the word THEN.

Line 3200 gets the next statement in sequence.

Line 3210 resets the pointer to its start.

Line 3220 checks to see if the statement was an ELSE part.

Line 3230 copies the statement into EXCESS\$. This means that the next time a source line is requested, EXCESS\$ will be used.

Line 3240 puts a label into the object code. This is where control will pass if the condition is FALSE.

Line 3250 exits.

Line 3260 is where control passes if the word ELSE did occur. It first increments the pointer past the word ELSE.

Line 3270 removes the spaces after the word ELSE.

Line 3280 checks to see if there was anything after the word ELSE.

Line 3290 creates a label for the THEN part of the statement to jump to.

Line 3300 inserts a line into the object code to make the code after the word THEN jump past the ELSE section. It also adds the label used when the expression was FALSE.

Line 3310 compiles the next statement.

Line 3320 inserts the ELSE label.

Line 3350 starts to define PROCexp which is used to compile an expression. The method used is the same as that in the BBC BASIC ROM.

Lines 3360 to 3750 carry out the same priority techniques as the ROM.

Line 3760 looks for GET as an operand, and assembles code to jump to the operating system routine OSRDCH. Notice that the escape flag is reset first, which effectively cancels the operation of the escape key. Notice also that RLT% and RLT\$ are set up to reflect the location of the result.

Line 3770 looks for a number.

Line 3780 checks that a number was found. Assuming that one was, RLT% is set up and the routine exits.

Line 3790 looks for a variable name.

Lines 3800 to 3830 set up RLT% to reflect the variable, assuming the variable was legal.

Lines 3850 to 3870 are used to push the contents of the primary accumulator onto the stack.

Lines 3890 to 3910 are used to assemble code to pull the secondary accumulator off the stack.

Line 3930 defines the procedure used to effect subtraction.

Line 3940 calls the procedure 'maths' to do the compilation. PROCmaths is also used by PROCadd. The parameters to PROCmaths are the assembly operations to be carried out. The first parameter is the one that must be used for the most significant bytes, whilst the second is the one used for least significant bytes. Obviously, the carry flag need only be set for the least significant part of the calculation.

Lines 3970 to 4090 are similar, except that they deal with the operations ADD, AND, EOR and OR.

Line 4130 defines PROCmaths, which is used for many different mathematical operations.

Line 4150 pushes the primary accumulator onto the stack if it is in use. This is because the second parameter is quite likely to use the accumulator, so it must be saved.

Line 4160 saves the results of the expression so far.

Line 4170 calls back to PROClevel__3 if the operation was ADD or SUBTRACT.

Line 4180 calls back to PROCexp if the operation was ORA or EOR.

Line 4190 calls back to PROClevel__1 if the operation was AND.

Line 4200 pulls the primary accumulator back into the secondary accumulator, if necessary.

Lines 4210 to 4250 deal with both arguments being numbers for the different operations that PROCmaths could carry out.

Lines 4260 to 4280 carry out the required operation for the other cases of RLT% and OLT%.

Lines 4300 to 4520 compile the various comparison operators. Notice that they use PROCcomparison.

Line 4540 starts to define PROCcomparison.

Line 4560 to 4580 are like the preamble to PROCmaths. The difference is that PROCcomparison then calls PROCdo__it, which moves the two parameters into the two accumulators, before assembling a JMP instruction to the address passed as a parameter.

Lines 4620 to 4680 are similar, except that they call the MULTIPLY routine to effect multiplication.

Lines 4700 to 4800 comprise PROCdo__it. This procedure simply moves the parameter indicated by OLT% and OLT\$ into the secondary accumulator, and the parameter indicated by RLT% and RLT\$ into the primary accumulator, before calling the relevant library routine.

Lines 4820 to 4870 deal with the unary minus operator. PROCoperand is called to get the argument to the minus operation, then the negating operation is carried out.

Lines 4890 to 4920 deal with brackets by simply calling PROCexp, and then verifying the presence of the closing right hand bracket.

Lines 4940 to 2970 deal with the unary byte indirection operator when it is used to interrogate a location.

Lines 4990 to 5020 deal with the word equivalent of the above.

Line 5040 starts the definition of the procedure which deals with the WRITE statement.

Line 5050 removes any spaces that may precede the first item in the list of things following WRITE.

Line 5060 checks to see if the first item is the null string. If it is, the end of the list has been reached, so the routine exits.

Line 5070 checks for a leading quote symbol, and branches off to line 5380 if one is found.

Line 5080 checks for the word TAB, and branches off to 5230 if the word is found.

Line 5090 checks for a comma, and branches off to 5200 if one is found.

Line 5100 checks for a single quote, and branches off to line 5170 if one is found.

Line 5110 ignores any semicolons found.

Line 5130 assumes that the item must be an expression, and calls PROCexp accordingly.

Lines 5140 and 5150 copy the data to be printed into IAC1, and insert a call to the print routine PNTNUM.

Line 5170 deals with the single quotation symbol. It assembles a call to OSNEWL.

Line 5180 moves the pointer past the quote.

Line 5200 deals with the comma by moving the pointer past it.

Line 5210 prints a space.

Line 5230 deals with the TAB keyword. It first checks for the opening bracket.

Line 5240 calls PROCexp to evaluate the first argument of the TAB section.

Line 5260 checks to see if the next character is a comma. If it is, this is a dual argument TAB, so it branches off to 5310.

Line 5270 checks for the closing bracket of the TAB.

Lines 5280 and 5290 assemble the required code. The technique is to *FX 134 to find the cursor position, and then alter the X position to reflect the indicated column, before moving the cursor, using VDU 31, to the new position. This technique allows you to write:

```
WRITE TAB(20);" A";TAB(10);" B"
```

This would have a different effect in BASIC.

Line 5310 deals with the dual argument TAB by using VDU 31. Only the X coordinate is set at this time, because it is the only one known.

Line 5320 increments the pointer past the comma.

Line 5330 calls PROCexp to evaluate the Y coordinate position.

Line 5350 checks for the closing bracket.

Line 5360 adds the second coordinate.

Line 5380 deals with quoted strings. It starts by inserting a call to the PNTSTNG routine, which prints the string following the JSR instruction, up to the first zero byte.

Line 5400 sets D\$ to the null string. D\$ will be used to build up the string to be printed.

Line 5410 checks for the line ending too quickly.

Line 5420 adds a new character to D\$.

Line 5430 continues this process until the right hand character of D\$ is a quote (we've reached the closing quote) and the length of D\$ is not 1 - in other words, the first time through the loop, D\$ will be CHR\$(34), which doesn't count.

Line 5440 places this string into the object program.

Line 5470 starts to define the procedure which deals with '!' when used to alter the contents of a memory location.

The coding of this procedure and the next, which deals with the byte equivalent, is similar to the maths routines we have already examined.

From the above description, you should be able to alter SLUG to your own requirements.

A useful technique to remember is that if you, say, wished to add the COLOUR statement, you could simply translate the COLOUR statement into 'OUTPUT 17,X', and let the existing code compile that. This could be done in a single line in PROCstatement.

Diversion: Studying DNA

It is unfortunately impossible to give a screen dump of the output of this program. It displays a beautiful, moving, pattern consisting of a column of DNA rising up the screen.

The program is capable of many effects, although the DNA effect is the most striking. Each depression of the space bar while the program is running will result in a new pattern.

> LIST

```

10 REM Filename:DNA
20
30 REM Simple DNA simulation
40
70 REM (c) 1982 Jeremy Ruston
80 MODE 4
90 VDU 23;8202;0;0;0;
100 VDU 19,0,7;0;19,1,0;0;
110 DIM X%(80),Y%(80)
120 REPEAT
130 CLS
140 VDU 29,640;824;
150 REPEAT
160 R% = RND(60) + 20
170 UNTIL R% MOD 2 = 1
180 FOR T% = 0 TO R%
190 X%(T%) = SIN(RAD(360*(T%/R%)))*200
200 Y%(T%) = COS(RAD(360*(T%/R%)))*200
210 NEXT T%
220 L% = 0
230 M% = 10
240 REPEAT

```

```

250 TIME = 0
260 MOVE X%(L%),Y%(L%)
270 L% = (L% + 1) MOD R%
280 PLOT 0,0,-32
290 DRAW X%(L%),Y%(L%)
300 MOVE X%(M%),Y%(M%)
310 M% = (M% + 1) MOD R%
320 PLOT 0,0,-32
330 DRAW X%(M%),Y%(M%)
340 DRAW X%(L%),Y%(L%)
350 VDU 11
360 REPEAT UNTIL TIME> 1
370 UNTIL INKEY(-99)
380 UNTIL FALSE

```

>

Diversion: More on Moire

The user interacts with this program to create Moire patterns. The results, as the screen dump shows, can resemble anything from a fan to an art deco organ.

In operation, pressing the shift key causes the pattern to progress horizontally. With no key pressed, the pattern proceeds diagonally. When the pattern reaches the bottom of the screen, it waits for a keypress. If return is pressed, the pattern is saved on disc/cassette, otherwise no further action is taken.

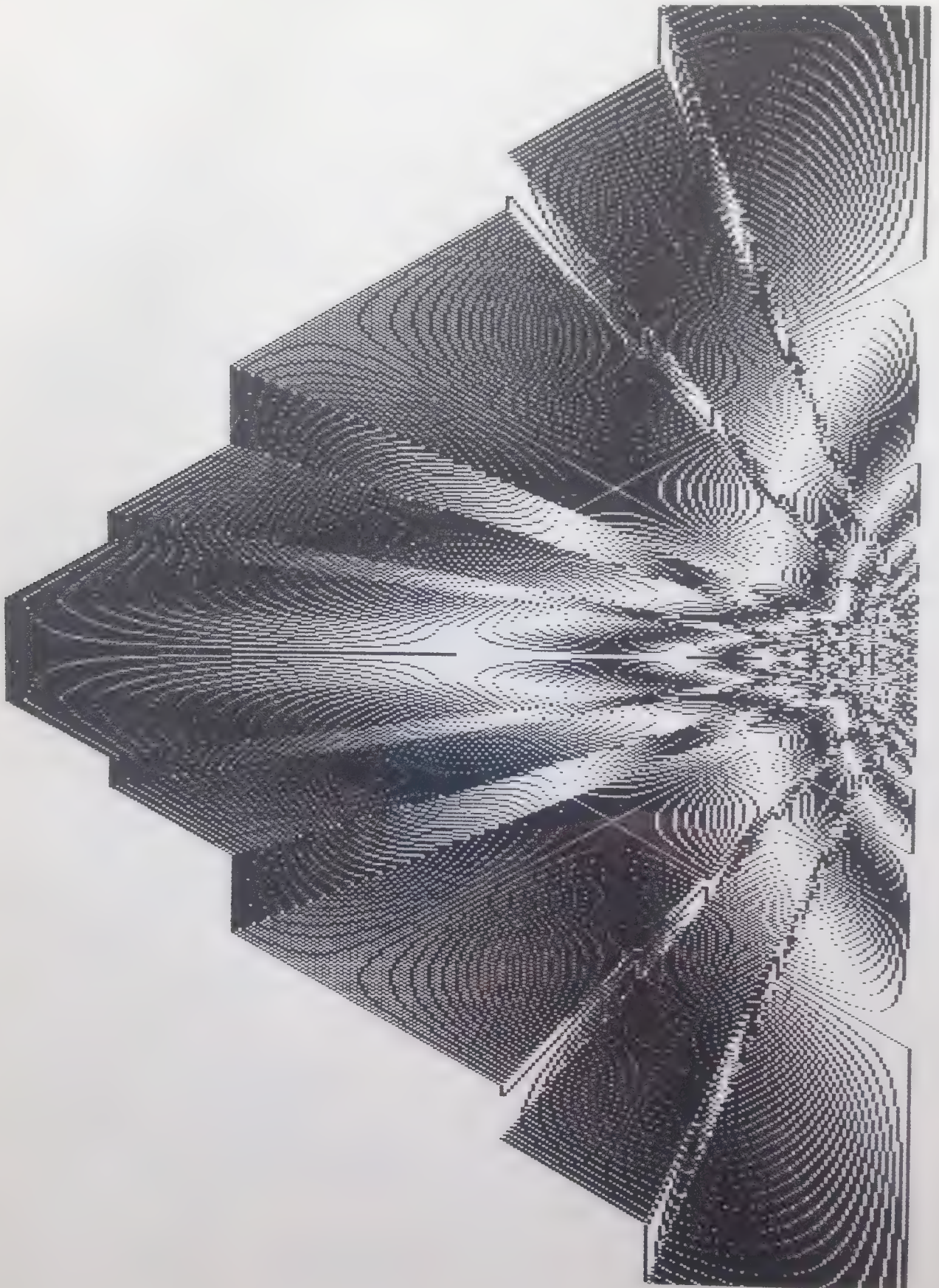
> LIST

```

10 REM Filename:FAN
20
30 REM Draw yourself a fan
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 0
80 X% = 639
90 Y% = 1023
100 REPEAT
110 MOVE 639,0
120 PLOT 6,X%,Y%
130 MOVE 640,0
140 PLOT 6,1279-X%,Y%
150 X% = X%-2
160 IF INKEY(-1) THEN Y% = Y%-4
170 UNTIL Y%< 0
180 IF GET = 13 THEN *SAVE P.FAN 3000 8000

```

>



Introduction to the BASIC ROM

This section describes both how the BASIC ROM was disassembled, and how the ROM works. The material presented here only applies directly to BASIC II.

It must be stressed that it does not matter which version of BASIC your machine is fitted with. Even if you have BASIC II, it would be foolish to use the information presented here to call routines in the ROM from your own programs. If you do, your programs will not operate with BASIC I or with some versions of the Tube. In addition, a machine code program is not always entered with the BASIC ROM switched into the sideways ROM space (this is true if you use *RUN with a disc filing system, for example). In fact, many of the more useful routines do not terminate with RTS instructions, but with JMPs to different areas. The exceptions are the floating point routines. However, if you wish to use floating point arithmetic, the chances are the program would be better written in BASIC. There is, of course, nothing stopping you from adapting routines from the ROM, but you should be sure you understand the routine in question before you attempt this.

However, the ROM is a good example of programming from which to improve your own. In addition, understanding how the ROM works enables you to make better use of BBC BASIC.

THE INTELLIGENT DISASSEMBLER

The disassembled listing of the ROM was obtained using an intelligent disassembler. The intelligence stems from the way it can work out which areas of the ROM are code and which are data, and display them accordingly.

The program used was as follows:

> LIST

```
10 REM Filename:ROMTRCE
20
30 REM Intelligent disassembler
40
50 REM (c) 1983 Jeremy Ruston
60
70 MODE 4
80
90 DATA 1brk2,2ora4,0***0,0***0,0***0,2ora1,2asl1,0***0
```



```

100 DATA 1php2,2ora0,1asl3,0***0,0***0,3ora1,3asl1,0***0
110 DATA 2bpl9,2ora5,0***0,0***0,0***0,2ora6,2asl6,0***0
120 DATA 1clc2,3ora7,0***0,0***0,0***0,3ora6,3asl6,0***0
130 DATA 3jsr1,2and4,0***0,0***0,2bit1,2and1,2rol1,0***0
140 DATA 1plp2,2and0,1rol3,0***0,3bit1,3and1,3rol1,0***0
150 DATA 2bmi9,2and5,0***0,0***0,0***0,2and6,2rol6,0***0
160 DATA 1sec2,3and7,0***0,0***0,0***0,3and6,3rol6,0***0
170 DATA 1rti2,2eor4,0***0,0***0,0***0,2eor1,2lsr1,0***0
180 DATA 1pha2,2eor0,1lsr3,0***0,3jmp1,3eor1,3lsr1,0***0
190 DATA 2bvc9,2eor5,0***0,0***0,0***0,2eor6,2lsr6,0***0
200 DATA 1cli2,3eor7,0***0,0***0,0***0,3eor6,3lsr6,0***0
210 DATA 1rts2,2adc4,0***0,0***0,0***0,2adc1,2ror1,0***0
220 DATA 1pla2,2adc0,1ror3,0***0,3jmp8,3adc1,3ror1,0***0
230 DATA 2bvs9,2adc5,0***0,0***0,0***0,2adc6,2ror6,0***0
240 DATA 1sei2,3adc7,0***0,0***0,0***0,3adc6,3ror6,0***0
250 DATA 0***0,2sta4,0***0,0***0,2sty1,2sta1,2stx1,0***0
260 DATA 1dey2,0***0,1txa2,0***0,3sty1,3sta1,3stx1,0***0
270 DATA 2bcc9,2sta5,0***0,0***0,2sty6,2sta6,2stx7,0***0
280 DATA 1tya2,3sta7,1txs2,0***0,0***0,3sta6,0***0,0***0
290 DATA 2ldy0,2lda4,2ldx0,0***0,2ldy1,2lda1,2ldx1,0***0
300 DATA 1tay2,2lda0,1tax2,0***0,3ldy1,3lda1,3ldx1,0***0
310 DATA 2bcs9,2lda5,0***0,0***0,2ldy6,2lda6,2ldx7,0***0
320 DATA 1clv2,3lda7,1txs2,0***0,3ldy6,3lda6,3ldx7,0***0
330 DATA 2cpy0,2cmp4,0***0,0***0,2cpy1,2cmp1,2dec1,0***0
340 DATA 1iny2,2cmp0,1dex2,0***0,3cpy1,3cmp1,3dex1,0***0
350 DATA 2bne9,2cmp5,0***0,0***0,0***0,2cmp6,2dec6,0***0
360 DATA 1cld2,3cmp7,0***0,0***0,0***0,3cmp6,3dec6,0***0
370 DATA 2cpx0,2sbc4,0***0,0***0,2cpx1,2sbc1,2inc1,0***0
380 DATA 1inx2,2sbc0,1nop2,0***0,3cpx1,3sbc1,3inc1,0***0
390 DATA 2beq9,2sbc5,0***0,0***0,0***0,2sbc6,2inc6,0***0
400 DATA 1sed2,3sbc7,0***0,0***0,0***0,3sbc6,3inc6,0***0
410
420 DIM A%255,B%255,C%255,D%255,E%255,SL%511,SH%511
430
440 FOR T% = 0 TO 255
450 READ A$
460 A%?T% = VAL(MID$(A$,1,1))
470 B%?T% = ASC(MID$(A$,2))
480 C%?T% = ASC(MID$(A$,3))
490 D%?T% = ASC(MID$(A$,4))
500 E%?T% = VAL(MID$(A$,5,1))
510 NEXT T%
520
530 PROCinit
540 REPEAT

```



```

550 PRINT CHR$30 ST%
560 IF ST% < > 0 THEN PROCfollow(FNpull)
570 UNTIL ST% = 0
580 PRINT "Completed."
590 PROCdissass
600 VDU 3
610 END
620
630 DEF PROCinit
640 LOCAL A%,P%,T%
650 VDU 28,0,16,39,0
660 ST% = 0
670 ROM% = &8000
680 P% = &8071
690 REPEAT
700 REPEAT
710 P% = P% + 1
720 UNTIL ?P% > = &80
730 T% = ?P%
740 A% = (T%?&8351)*256 + (T%?&82DF)
750 IF A% > &8000 THEN PROCpush(A% MOD &4000)
760 P% = P% + 2
770 UNTIL P% > &836C
780 PROCpush(&B402-&8000)
790 PROCpush(0)
800 ENDPROC
810
820 DEF PROCpush(A%)
830 IF FNtest(A%) THEN ENDPROC
840 IF ST% = 512 THEN PRINT "Stack overflow":END
850 SL%?ST% = A%
860 SH%?ST% = A% DIV 256
870 ST% = ST% + 1
880 ENDPROC
890
900 DEF FNpull
910 IF ST% = 0 THEN PRINT "Stack underflow":END
920 ST% = ST% - 1
930 = (SL%?ST%) + (SH%?ST%)*256
940
950 DEF PROCset(A%)
960 ?(&7000 + A% DIV 8) = ?(&7000 + A% DIV 8) OR 2^(7-(A% MOD
8))
970 ENDPROC
980

```

```

990 DEF FNtest(A%) = ?(&7000 + A% DIV 8) AND 2^(7-(A% MOD
8))
1000
1010 DEF PROCfollow(PNTR%)
1020 IF FNtest(PNTR%) OR PNTR% > = &4000 OR SGN(PNTR%) = -
1 THEN ENDPROC
1030 PROCinspect
1040 GOTO 1020
1050
1060 DEF PROCinspect
1070 LOCAL INS%,L%,T%,AD%
1080 INS% = PNTR%?ROM%
1090 L% = INS%?A%
1100 IF L% = 0 THEN PNTR% = -10:ENDPROC
1110 FOR T% = PNTR% TO PNTR% + L%-1
1120 PROCset(T%)
1130 NEXT T%
1140 AD% = !(ROM% + PNTR% + 1) AND &FFFF
1150 IF INS% = &20 THEN PROCpush(AD%-ROM%)
1160 IF INS% = &4C THEN PROCpush(AD%-ROM%):PNTR% = -10
1170 IF INS% = &6C OR INS% = &40 OR INS% = &60 THEN PNTR
% = -10
1180 AD% = ?(ROM% + PNTR% + 1)
1190 IF AD% < 128 THEN AD% = AD% + 2 + PNTR% ELSE AD% =
AD%-254 + PNTR%
1200 IF ((INS%-16) MOD 32) = 0 THEN PROCpush(AD%)
1210 PNTR% = PNTR% + L%
1220 IF INS% = 0 THEN PNTR% = -1
1230 ENDPROC
1240
1250 DEF PROCdissass
1260 CLS
1270 PRINT "" "Press 'P' for printer output..."
1280 A$ = GET$
1290 IF A$ = "P" OR A$ = "p" THEN VDU 2
1300 PNTR% = 0
1310 IF FNtest(PNTR%) THEN PROCdis ELSE PROCdata
1320 PRINT
1330 REM PRINT STRING$(50-POS,"__")
1340 IF PNTR% = &4000 THEN ENDPROC
1350 GOTO 1310
1360
1370 DEF PROCdis
1380 LOCAL INS%,L%,M%,DT%
1390 INS% = PNTR%?ROM%

```

```

1400 PRINT ;~PNTR% + ROM%";":";CHR$(B%?INS%);CHR$(C%?
INS%);CHR$(D%?INS%);
1410 L% = INS%?A%
1420 IF L% = 0 THEN PRINT "A totally impossible error has
occured":END
1430 IF L% = 2 THEN DT% =?(PNTR% + ROM% + 1)
1440 IF L% = 3 THEN DT% =!(PNTR% + ROM% + 1) AND &FFFF
1450 M% = INS%?E%
1460 IF M% = 0 THEN PRINT " #&";~DT%;
1470 IF M% = 1 THEN PRINT " &";~DT%;
1480 IF M% = 2 THEN
1490 IF M% = 3 THEN PRINT " A";
1500 IF M% = 4 THEN PRINT " (&";~DT%;",X)";
1510 IF M% = 5 THEN PRINT " (&";~DT%;"),Y";
1520 IF M% = 6 THEN PRINT " &";~DT%;",X";
1530 IF M% = 7 THEN PRINT " &";~DT%;",Y";
1540 IF M% = 8 THEN PRINT " (&";~DT%;")";
1550 IF M% = 9 AND DT% < 128 THEN PRINT " &";~PNTR% +
ROM% + DT% + 2;
1560 IF M% = 9 AND DT% > 127 THEN PRINT " &";~PNTR% +
ROM% + DT% - 254;
1570 PNTR% = PNTR% + L%
1580 ENDPROC
1590
1600 DEF PROCdata
1610 LOCAL CH%
1620 PRINT ;~PNTR% + ROM%";":";RIGHT$("0" + STR$~(PNTR%?
ROM%),2);"-";
1630 CH% = PNTR%?ROM% MOD 128
1640 IF CH% > 31 AND CH% < 127 THEN VDU CH%
1650 PNTR% = PNTR% + 1
1660 ENDPROC
>

```

The program first works out which areas of the ROM are data areas and which areas contain executable code. To do this, it maintains a map of the ROM in the MODE 4 screen. Using this map, each of 16384 pixels represents a different byte of the ROM. A set pixel indicates a byte that is part of the code and an unset pixel indicates a data area. Once this map has been built up, it is a simple matter to scan through the ROM, disassembling or dumping each byte according to the table.

To generate the table, the program pushes every entry point of the ROM onto a stack. The entry points are the normal &8000 entry, the error vector address and the action address of each keyword (which will be explained later). Once the stack has been set up, the program carries out the following tasks:

- 1 Exit if the stack is empty
- 2 Pull an address off the stack
- 3 Set the pixels corresponding to the instruction. Thus, one pixel will need to be set for a single byte instruction, two for a double byte instruction and three for a triple byte instruction
- 4 If the instruction is a branch instruction, a jump instruction or a subroutine call, push the destination address on the stack
- 5 If the instruction is an RTS, BRK or JMP, go back to step 1
- 6 Move to the next instruction
- 7 Go back to step 3

The program can easily be adapted to become a normal disassembler or to disassemble the operating system or any other program.

Line by line notes:

Lines 90 to 400 contain data about each 6502 instruction. Each instruction name is flanked on the left by the length of the instruction, and on the right by the addressing mode of the instruction. The addressing modes are assigned as follows:

- 0 Immediate addressing
- 1 Absolute addressing
- 2 Implied addressing
- 3 Accumulator addressing
- 4 Pre-indexed with X indirect addressing
- 5 Post-indexed with Y indirect addressing
- 6 Indexed with X addressing
- 7 Indexed with Y addressing
- 8 Indirect addressing
- 9 Relative addressing

Line 420 dimensions byte arrays to hold the above data. The arrays are used as follows:

- A% Instruction length
- B% First letter of instruction
- C% Second letter of instruction
- D% Third letter of instruction
- E% Addressing mode
- SL%/SH% Stack

Lines 440 to 510 fill the arrays A%, B%, C%, D% and E% from the data statements.

Line 530 calls PROCinit, which sets up the stack.

Line 540 starts the main REPEAT loop of the program. This part of the program loops through all the addresses on the stack, setting a flag for each one executed. The array of flags (held on screen) is used to decide which bytes are data and which are instructions.

Line 550 prints the stack pointer at the top of the screen. This allows the progress of the program to be checked, since the program will have completed the array of flags only when the stack depth reaches zero.

Line 560 ensures the stack is not empty and then starts to trace the address at the top of the stack.

Line 570 terminates the loop when the stack is empty.

Line 580 informs the user that the first process has been completed.

Line 590 calls PROCdissass to disassemble the ROM.

Line 600 turns off the printer, which could have been activated in PROCdissass.

Line 610 ends the program.

Line 630 starts to define PROCinit.

Line 650 sets up a text window at the top of the screen. This window is used to ensure that the lower area of the screen, which is used for the map of those locations visited, is not disturbed through scrolling.

Line 660 resets the stack pointer.

Line 670 sets the start address of the ROM to &8000.

Line 680 starts to loop through the keyword table, pushing the action addresses of the keywords onto the stack. The action addresses will be detailed in due course. The process is initiated by setting P% to the first letter of the first keyword.

Line 690 actually starts the loop.

Line 700 starts a further loop, this time through the letters of the keyword.

Line 710 increments P% to the next letter.

Line 720 stops this process when P% points to the token.

Line 730 gets the token value.

Line 740 extracts the action address of the keyword.

Line 750 verifies that the address is legal, and pushes its offset from the start of the ROM onto the stack.

Line 760 increments P% to the first letter of the next keyword.

Line 770 terminates this process when P% passes the final keyword.

Line 780 pushes the address of the error handling routine onto the stack. This address is the one normally contained in &202,&203.

Line 790 pushes the start address of the ROM onto the stack. Again, this is offset from the first location of the ROM.

Line 800 terminates PROCinit.

Lines 820 to 880 push a given value onto the stack. FNtest is used to prevent the address being pushed if the address in question has already been examined. FNtest will return TRUE for a location that has already been examined, and FALSE otherwise.

Lines 900 to 930 pull a value from the stack.

Line 950 starts to define PROCset, which sets the flag corresponding to a particular location. A single bit of memory from &7000 onwards is assigned to each byte of the ROM.

Line 960 works out the address of the byte containing the bit to be set by dividing the offset by eight, and adding &7000. The relevant bit at this location is then set.

Line 970 terminates the procedure.

Line 990 tests the status of a particular bit. The same formula is used as in PROCset.

Line 1010 starts the definition of PROCfollow. This procedure simulates the execution of the instructions from PNTR% onwards.

Line 1020 exits the procedure if PNTR% points to a location that has been previously examined, or PNTR% has moved out of range.

Line 1030 inspects the location pointed to. This action will alter PNTR% to point to the next location to be examined.

Line 1040 loops back to repeat the process.

Line 1060 defines PROCinspect.

Line 1080 extracts the instruction code of the current instruction.

Line 1090 extracts the length of the current instruction.

Line 1100 exits the routine if the length is given as zero. This means that an illegal instruction has been encountered. PNTR% is set to a degenerate value to ensure that a new address is popped off the stack.

Lines 1110 to 1130 set the bits corresponding the current instruction.

Line 1140 extracts the two byte address following the instruction.

Line 1150 checks for a JSR instruction. If one is present, it pushes the destination of the call onto the stack.

Line 1160 checks for a direct JMP instruction. If present, the destination address is pushed, and PNTR% is set to a degenerate value, since there is no guarantee that the next instruction will be executed.

Line 1170 checks for indirect jump instructions and RTS and RTI. These instructions simply result in PNTR% being set to a degenerate value.

Line 1180 extracts the single byte of data following the current instruction.

Line 1190 converts this relative offset into an absolute address.

Line 1200 identifies branch instructions, and pushes their destination onto the stack.

Line 1210 increments PNTR% to the next instruction.

Line 1220 checks for the BRK instruction, and sets PNTR% to a degenerate value if it is found.

Line 1230 terminates the routine.

Line 1250 starts to define PROCdisass.

Lines 1260 to 1290 start the printer if required. Without printer output, the output of the program is difficult to digest.

Line 1300 sets PNTR% to zero. This is used as a pointer to the next byte to be dealt with.

Line 1310 works out if PNTR% points to an instruction or to data, and acts accordingly.

Line 1320 moves to a new line after the instruction code.

Line 1330 is an optional statement that follows each line of disassembly with a row of underscore characters, to facilitate commenting of the output code.

Line 1340 ends the routine when the end of the ROM is reached.

Line 1350 loops back for the next byte.

Line 1370 starts to define PROCdissass, which disassembles the single instruction pointed to by PNTR%.

Line 1390 extracts the instruction's op-code.

Line 1400 prints the current address, followed by the instruction code.

Line 1410 extracts the length of the instruction.

Line 1420 flags an error which is unlikely to occur.

Lines 1430 and 1440 set the variable DT% to the data of the instruction. The setting is different depending upon the length of the instruction.

Line 1450 extracts the addressing mode of the instruction.

Lines 1460 to 1560 decode this information.

Line 1570 advances the pointer to the next instruction.

Line 1600 starts to define PROCdata.

Line 1620 prints the address of the data, followed by the data in hexadecimal.

Lines 1630 and 1640 print the data as a character if this is possible.

Line 1650 advances the pointer to the next instruction.

Line 1660 terminates the routine.

MEMORY USAGE

The BASIC ROM makes the following use of memory:

00-LSB of LOMEM

01-MSB of LOMEM

02-LSB of VARTOP. VARTOP points to the first location above the current program that is not being used for storing variables. Thus, the next variable to be defined will be stored at VARTOP

03-MSB of VARTOP

- 04-LSB of stack pointer. The BASIC stack resides just below HIMEM. It is used for expression evaluation and handling functions and procedures
- 05-MSB of stack pointer
- 06-LSB of HIMEM
- 07-MSB of HIMEM
- 08-LSB of ERL
- 09-MSB of ERL
- 0A-Offset for PTR #1. This pointer is accessed by instructions of the form 'LDA (&0B),Y'. It normally points to the next byte of the BASIC program to be examined
- 0B-LSB of PTR #1
- 0C-MSB of PTR #1
- 0D-RND store. These locations hold the last random number generated, which is used to generate the next one
- 0E-RND store
- 0F-RND store
- 10-RND store
- 11-RND store
- 12-LSB of TOP
- 13-MSB of TOP
- 14-Print field width. This location has the value of @% copied into it before a PRINT statement
- 15-Hex/decimal output. If bit 7 is set, all printing will occur in hexadecimal, else decimal is used. This location is reset to zero when the PRINT statement is entered, so changing it has no lasting effect
- 16-LSB error vector. These locations point to a fragment of a BASIC program used to handle errors. The address following the word 'ERROR' in 'ON ERROR...' is put into these locations.
- 17-MSB error vector
- 18-MSB of PAGE. The LSB of page does not need to be stored, since it is always zero
- 19-LSB of PTR #2. PTR #2 is like PTR #1, but is mainly used for evaluating expressions.
- 1A-MSB of PTR #2
- 1B-Offset for PTR #2
- 1C-DATA pointer. This points to the character after the last DATA item read
- 1D-DATA pointer
- 1E-COUNT
- 1F-LISTO option
- 20-TRACE flag. If this location contains &FF, TRACE will occur whenever a new line is executed.
- 21-LSB maximum TRACE number
- 22-MSB maximum TRACE number
- 23-WIDTH
- 24-REPEAT stack pointer. The BASIC loop statements maintain their own stacks in page 5, for speed.
- 25-GOSUB stack pointer

26-FOR stack pointer

27-Variable type. The type of results to expressions are stored here. The codes used are:

00-String

40-Integer

FF-Real

28-OPT value

29-Various purposes

2A-Integer accumulator (IAC). These two locations sometimes holds the address of the value of a variable

2B-IAC

2C-IAC. This location sometimes holds the type of a variable name. The codes used are:

00-8 bit byte

04-32 bit integer variable

05-40 bit floating point variable

80-A string at a defined address

81-A dynamic string

2D-IAC

2E-FAC # 1 sign

2F-FAC # 1 over/underflow

30-FAC # 1 exponent

31-FAC # 1 mantissa

32-FAC # 1 mantissa

33-FAC # 1 mantissa

34-FAC # 1 mantissa

35-FAC # 1 rounding byte

36-String length. This byte contains the length of the string stored from location &600 onwards

37-LSB pointer

38-MSB pointer

39-Various purposes

3A-Various purposes

3B-FAC # 2 sign

3C-FAC # 2 over/underflow

3D-FAC # 2 exponent

3E-FAC # 2 mantissa

3F-FAC # 2 mantissa

40-FAC # 2 mantissa

41-FAC # 2 mantissa

42-FAC # 2 rounding byte

43 to 4E-Various purposes

4F to 8F-Free for the user

Many locations are used for more than one purpose, but the comments in the ROM listing remove any ambiguity.

The other locations used are as follows:

400-46B	The resident integer variables. The variables @% to Z% are each assigned four bytes
46C-47F	Temporary floating point workspace
480-4F9	Variable catalogue
500-595	FOR stack. There is room on the stack for 10 fifteen byte entries. Each entry has the following format:
0-1	Address of the loop variable
2	Type of the loop variable. The type is in the same format as the CALL statement
3-7	Step size. For integer loops, only four bytes are used
8-C	Terminating value. Again, only four bytes are used for integer loops
D-E	Address of the statement after the FOR statement
5A4-5B7	The REPEAT stack. This table holds the LSB of the address of the character after the REPEAT of the current loop
5B8-5CB	The MSB REPEAT stack
5CC-5E5	The GOSUB stack. This table holds the LSB of the address of the character after the GOSUB of the current subroutine call
5E6-5FF	The MSB GOSUB stack
600-6FF	The string buffer. This area holds strings as they are processed
700-7FF	The keyboard buffer. This area is used to hold the commands you type in at the keyboard

VARIABLE STORAGE

Variables, apart from the resident integer variables, are stored in two areas; the variable catalogue and the area from LOMEM to VARTOP.

The variable catalogue contains 59 two byte entries. Each entry gives the address of the first variable with a given first letter. If no variable exists with a given first letter, the entry is zero. For example,

&482,3 points to the first variable starting with 'A'
 *484,5 points to the first variable starting with 'B'
 *4F4,5 points to the first variable starting with 'z'

In addition, &4F6,7 points to the first FN block and &4F8,9 points to the first PROC block.

Variables and procedures are stored in variable and PROC/FN information blocks. A variable information block has the format:

Address of the next variable with the same initial letter (2 bytes; with the MSB set to zero if no other variable with the same initial letter exists).

Name of variable (including '\$', '%' or '(', but without the first letter, which is inherent)

Zero

Value

The 'value' section has a different meaning for each type of variable. PROC/FN information blocks are stored in the same format, except that the entire name of the PROC/FN is stored.

For floating point variables, the value is simply the packed five byte floating point representation of the present value of the variable. Integer variables are stored using four bytes in the normal way. Strings are stored in four bytes. The first two give the address of the string, the third gives the number of bytes allocated to the string and the fourth gives the current length of the string. The contents of the string are stored separately in the same area. The value of a PROC/FN information block is simply the address of the PROC/FN.

Arrays are stored as a list of values, preceded by an area giving details of the number of dimensions in the array. This is stored as $2*n+1$ (where 'n' gives the number of dimensions of the array) followed by the two byte values of each subscript.

LINE NUMBERS

Line numbers are also stored in a special way inside the text of the program. The line is encoded as a token of &8D followed by three bytes giving the line number encoded thus:

Bits —	7	6	5	4	3	2	1	0
Byte 1 —	0	1	128s	64s	0	16384s	0	0
Byte 2 —	0	1	32s	16s	8s	4s	2s	1s
Byte 3 —	0	1	8192s	4096s	2048s	1024s	512s	256s

see p334

Thus, a modified binary form is used. This done so that line numbers are of constant length in bytes, making the renumbering process easier. Straight binary has not been used, because the interpreter would crash if a line contained a byte less than 32.

THE KEYWORD TABLE

The following program prints out information about each BASIC keyword. The program cannot easily be explained until the ROM itself is examined.

The information printed is the name of the keyword, followed by its token, the attribute bits and finally the action address of the keyword.

The attribute bits are explained in the ROM listing itself.

The action address section is useful if you are interested in how a particular keyword is implemented, since you can look up the address in this table, and find the keyword's code in the ROM directly.


```

> RUN
AND .....80.....00000000.....044C
ABS.....94.....00000000.....AD6A
ACS.....95.....00000000.....A8D4
ADVAL.....96.....00000000.....AB33
ASC.....97.....00000000.....AC9E
ASN.....98.....00000000.....A8DA
ATN.....99.....00000000.....A907
AUTO.....C6.....00010000.....90AC
BGET.....9A.....00000001.....BF6F
BPUT.....D5.....00000011.....BF58
COLOUR.....FB.....00000010.....938E
CALL.....D6.....00000010.....8ED2
CHAIN.....D7.....00000010.....BF2A
CHR$.....BD.....00000000.....B3BD
CLEAR.....D8.....00000001.....928D
CLOSE.....D9.....00000011.....BF99
CLG.....DA.....00000001.....8EBD
CLS.....DB.....00000001.....8EC4
COS.....9B.....00000000.....A98D
COUNT.....9C.....00000001.....AEF7
DATA.....DC.....00100000.....8B7D
DEG.....9D.....00000000.....ABC2
DEF.....DD.....00000000.....8B7D
DELETE.....C7.....00010000.....8F31
DIV.....81.....00000000.....1F4F
DIM.....DE.....00000010.....912F
DRAW.....DF.....00000010.....93E8
ENDPROC.....E1.....00000001.....9356
END.....E0.....00000001.....8AC8
ENVELOPE.....E2.....00000010.....B472
ELSE.....8B.....00010100.....B14D
EVAL.....A0.....00000000.....ABE9
ERL.....9E.....00000001.....AF9F
ERROR.....85.....00000100.....E6D2
EOF.....C5.....00000001.....ACB8
EOR.....82.....00000000.....7D4D
ERR.....9F.....00000001.....AFA6
EXP.....A1.....00000000.....AA91
EXT.....A2.....00000001.....BF46
FOR.....E3.....00000010.....B7C4
FALSE.....A3.....00000001.....AECA
FN.....A4.....00001000.....B195
GOTO.....E5.....00010010.....B8CC
GET$.....BE.....00000000.....AFBF

```

GET.....	A5.....	00000000.....	AFB9
GOSUB.....	E4.....	00010010.....	B888
GCOL.....	E6.....	00000010.....	937A
HIMEM.....	93.....	01000011.....	AF03
INPUT.....	E8.....	00000010.....	BA44
IF.....	E7.....	00000010.....	98C2
INKEY\$.....	BF.....	00000000.....	B026
INKEY.....	A6.....	00000000.....	ACAD
INT.....	A8.....	00000000.....	AC78
INSTR(.....	A7.....	00000000.....	ACE2
LIST.....	C9.....	00010000.....	B59C
LINE.....	86.....	00000000.....	B600
LOAD.....	C8.....	00000010.....	BF24
LOMEM.....	92.....	01000011.....	AEFC
LOCAL.....	EA.....	00000010.....	9323
LEFT\$(.....	C0.....	00000000.....	AFCC
LEN.....	A9.....	00000000.....	AED1
LET.....	E9.....	00000100.....	8BE4
LOG.....	AB.....	00000000.....	ABA8
LN.....	AA.....	00000000.....	A7FE
MID\$(.....	C1.....	00000000.....	B039
MODE.....	EB.....	00000010.....	939A
MOD.....	83.....	00000000.....	E445
MOVE.....	EC.....	00000010.....	93E4
NEXT.....	ED.....	00000010.....	B695
NEW.....	CA.....	00000001.....	8ADA
NOT.....	AC.....	00000000.....	ACD1
OLD.....	CB.....	00000001.....	8AB6
ON.....	EE.....	00000010.....	B915
OFF.....	87.....	00000000.....	1148
OR.....	84.....	00000000.....	E44D
OPENIN.....	8E.....	00000000.....	BF78
OPENOUT.....	AE.....	00000000.....	BF7C
OPENUP.....	AD.....	00000000.....	BF80
OSCLI.....	FF.....	00000010.....	BEC2
PRINT.....	F1.....	00000010.....	8D9A
PAGE.....	90.....	01000011.....	AEC0
PTR.....	8F.....	01000011.....	BF47
PI.....	AF.....	00000001.....	ABCB
PLOT.....	F0.....	00000010.....	93F1
POINT(.....	B0.....	00000000.....	AB41
PROC.....	F2.....	00001010.....	9304
POS.....	B1.....	00000001.....	AB6D
RETURN.....	F8.....	00000001.....	B8B6
REPEAT.....	F5.....	00000000.....	BBE4

```

REPORT.....F6.....00000001.....BFE4
READ.....F3.....00000010.....BB1F
REM.....F4.....00100000.....8B7D
RUN.....F9.....00000001.....BD11
RAD.....B2.....00000000.....ABB1
RESTORE.....F7.....00010010.....BAE6
RIGHT$(.....C2.....00000000.....AFEE
RND.....B3.....00000001.....AF49
RENUMBER...CC.....00010000.....8FA3
STEP.....88.....00000000.....D049
SAVE.....CD.....00000010.....BEF3
SGN.....B4.....00000000.....AB88
SIN.....B5.....00000000.....A998
SQR.....B6.....00000000.....A7B4
SPC.....89.....00000000.....8E4D
STR$.....C3.....00000000.....B094
STRING$(.....C4.....00000000.....B0C2
SOUND.....D4.....00000010.....B44C
STOP.....FA.....00000001.....8AD0
TAN.....B7.....00000000.....A6BE
THEN.....8C.....00010100.....A0D3
TO.....B8.....00000000.....AEDC
TAB(.....8A.....00000000.....9545
TRACE.....FC.....00010010.....9295
TIME.....91.....01000011.....AEB4
TRUE.....B9.....00000001.....ACC4
UNTIL.....FD.....00000010.....BBB1
USR.....BA.....00000000.....ABD2
VDU.....EF.....00000010.....942F
VAL.....BB.....00000000.....AC2F
VPOS.....BC.....00000001.....AB76
WIDTH.....FE.....00000010.....B4A0
PAGE.....D0.....00000000.....9283
PTR.....CF.....00000000.....BF30
TIME.....D1.....00000000.....92C9
LOMEM.....D2.....00000000.....926F
HIMEM.....D3.....00000000.....925D

```

> LIST

10 REM Filename:TOKENS

20

30 REM Token information

40

50 REM (c) 1983 Jeremy Ruston

60

70 LOC% = &8071


```

80 REPEAT
90 REPEAT
100 VDU ?LOC%
110 LOC% = LOC% + 1
120 UNTIL ?LOC% AND 128
130 PRINT STRING$(12-POS, ".");
140 TKN% = ?LOC%
150 TYP% = LOC%?1
160 LOC% = LOC% + 2
170 PRINT RIGHT$("0" + STR$__(TKN%),2); "...";
180 MSK% = 128
190 FOR BIT% = 0 TO 7
200 IF (TYP% AND MSK%) THEN PRINT "1"; ELSE PRINT "0";
210 MSK% = MSK% DIV 2
220 NEXT BIT%
230 PRINT "...";RIGHT$("0000" + STR$__(TKN%?&8351*256 + TKN
%?&82DF),4)
240 UNTIL TKN% = &D3
>

```

ALPHABETICAL ROM INDEX

This is an alphabetical listing of the routines in the ROM:

! binary operator.....	967F
! unary operator.....	95A5
\$ unary operator.....	95B0
* operator routine.....	9D3C
* statement routine.....	8B73
+ operator routine.....	9C4E
- operator routine.....	9CB5
- unary operator.....	AD8C
/ operator routine.....	9DE5
< operator routine.....	9BC0
< = operator routine.....	9BD4
< > operator routine.....	9BDF
= operator routine.....	9BAE
= statement routine.....	8B47
> operator routine.....	9BE8
> = operator routine.....	9BFA
? binary operator.....	967B
? unary operator.....	95A7
ABS function routine.....	AD6A

ACS function routine.....	A8D4
ADVAL function routine.....	AB33
AND operator routine.....	9B7A
ASC function routine.....	AC9E
ASN function routine.....	A8DA
ATN function routine.....	A907
AUTO command routine.....	90AC
Assemble a single instruction.....	85BA
Assembler base op-codes.....	84C5
Assembler entry.....	8504
Assembler exit.....	84FD
Assembler label definition.....	85A5
Assembler mnemonics table (LSB).....	8451
Assembler mnemonics table (MSB).....	848B
Assign value to numeric variable.....	B4B1
BASIC error handler.....	B433
BGET statement routine.....	BF6F
BPUT statement routine.....	BF58
BRK handling routine.....	B402
CALL statement routine.....	8ED2
CHAIN statement routine.....	BF2A
CHR\$ function routine.....	B3BD
CLEAR statement routine.....	928D
CLG statement routine.....	8EBD
CLOSE statement routine.....	BF99
CLS statement routine.....	8EC4
COLOUR statement routine.....	938E
COS function routine.....	A98D
COUNT function routine.....	AEF7
Check FAC#1.....	A1DA
Check for a 'Bad program'.....	BE6F
Check for a comma at PTR#2.....	8AAE
Check for end of statement.....	9857
Check if A is alphanumeric.....	8926
Clear a new variable.....	9531
Clear stacks.....	BD3A
Clear variables.....	BD20
Cold start.....	8ADD
Compare two items.....	9A9E
Convert a line number to binary.....	889C
Convert number to a string.....	9EDF
Convert string to a number.....	A07B
Create a PROC/FN catalogue entry.....	94ED
Create a catalogue entry.....	94FC
DATA statement routine.....	8B7D

DEF statement routine.....	8B7D
DEG function routine.....	ABC2
DELETE command routine.....	8F31
DIM statement routine.....	912F
DIV operator routine.....	9E0A
DRAW statement routine.....	93E8
Decode hexadecimal number.....	AE6D
Decrement MAN#1.....	A4C7
Delete a line from program.....	BC2D
Delete bytes in buffer.....	887C
Discard the IAC from stack.....	BDFE
END statement routine.....	8AC8
ENDPROC statement routine.....	9356
ENVELOPE statement routine.....	B472
EOF function routine.....	ACB8
EOR operator routine.....	9B55
EQUX in assembler.....	883A
ERL function routine.....	AF9F
ERR function routine.....	AFA6
EVAL function routine.....	ABE9
EXP function routine.....	AA91
EXT function routine.....	BF46
Error handler.....	B402
Evaluate expression.....	9B29
Evaluate integer expression.....	8821
Evaluate operand.....	ADEC
Exchange FAC#1 and (&4B).....	A4D6
Exponentiation routine.....	9E35
Expression evaluator (integer).....	8821
FALSE function routine.....	AECA
FN function routine.....	B195
FOR statement routine.....	B7C4
Find ERL.....	B3C5
Fix FAC#1.....	A3FE
Floating point addition.....	9C8B
Floating point comparison.....	9A50
Floating point multiplication.....	9D20
Floating point subtraction.....	9CE1
GCOL statement routine.....	937A
GET function routine.....	AFB9
GET\$ function routine.....	AFBF
GOSUB statement routine.....	B888
GOTO statement routine.....	B8CC
Get a variable name.....	95DD
Get an array/PROC/FN name.....	9559

Get character from PTR#1.....	8A97
Get character from PTR#2.....	8A8C
Get file handle.....	BFA9
Get the value of a variable.....	B32C
HIMEM function routine.....	AF03
HIMEM statement routine.....	925D
IF statement routine.....	98C2
INKEY function routine.....	ACAD
INKEY\$ function routine.....	B026
INPUT statement routine.....	BA44
INPUT# statement routine.....	B9CF
INSTR function routine.....	ACE2
INT function routine.....	AC78
Increment MAN#1.....	A4B6
Increment the IAC.....	9222
Initialisation of language.....	8023
Insert a line into program.....	BC8D
Integer addition.....	9C53
Integer comparison.....	9AA2
Integer division.....	99BE
Integer multiplication.....	9D41
Integer subtraction.....	9CBA
Keyword action addresses (LSB).....	836D
Keyword action addresses (MSB).....	83E0
Keyword table.....	8071
LEFT\$ function routine.....	AFCC
LEN function routine.....	AED1
LET statement routine.....	8BE4
LEVEL 1 of expression evaluator.....	9B29
LEVEL 2 of expression evaluator.....	9B72
LEVEL 3 of expression evaluator.....	9B9C
LEVEL 4 of expression evaluator.....	9C42
LEVEL 5 of expression evaluator.....	9DD1
LEVEL 6 of expression evaluator.....	9E20
LIST command routine.....	B59C
LISTO command routine.....	B58A
LN function routine.....	A7FE
LOAD command routine.....	BF24
LOCAL statement routine.....	9323
LOG function routine.....	ABA8
LOMEM function routine.....	AEFC
LOMEM statement routine.....	926F
Label definition in assembler.....	85A5
Line number decoding.....	97EB
Line number encryption.....	88F3

Line numbers; convert to binary.....	889C
Line numbers; tokenise.....	88DB
Load a new program.....	BE62
MID\$ function routine.....	B039
MOD operator routine.....	9E01
MODE statement routine.....	939A
MOVE statement routine.....	93E4
Mnemonic table (LSB).....	8451
Mnemonic table (MSB).....	848B
Move to next statement.....	9880
NEW command routine.....	8ADA
NEXT statement routine.....	B695
NOT function routine.....	ACD1
Normalise FAC#1.....	A303
OLD command routine.....	8AB6
ON ERROR OFF statement routine.....	B8E4
ON ERROR statement routine.....	B8F2
ON statement routine.....	B915
OPENIN function routine.....	BF78
OPENOUT function routine.....	BF7C
OPENUP function routine.....	BF80
OR operator routine.....	9B3A
OSCLI statement routine.....	BEC2
Op-code base table.....	84C5
PAGE function routine.....	AEC0
PAGE statement routine.....	9283
PI function routine.....	ABCB
PLOT statement routine.....	93F1
POINT function routine.....	AB41
POS function routine.....	AB6D
PRINT statement routine.....	8D9A
PRINT# statement routine.....	8D2B
PROC statement routine.....	9304
PTR function routine.....	BF47
PTR statement routine.....	BF30
Pack FAC#1 to &46C onwards.....	A385
Pack FAC#1 to &471 onwards.....	A37D
Pack FAC#1 to &476 onwards.....	A381
Pack FAC#1 to (&4B).....	A38D
Parameters; unstacking.....	8CC1
Print A in hexadecimal.....	B545
Print a character.....	B50E
Print a string.....	BFCF
Print number in decimal.....	9EDF
Print number in hexadecimal.....	9E90

Print the IAC.....	991F
Pull a string from stack.....	BDCB
Pull the IAC from stack.....	BDEA
Push FAC#1 onto stack.....	BD51
Push a string onto stack.....	BDB2
Push anything onto stack.....	BD90
Push the IAC onto stack.....	BD94
RAD function routine.....	ABB1
READ statement routine.....	BB1F
REM statement routine.....	8B7D
RENUMBER command routine.....	8FA3
REPEAT statement routine.....	BBE4
REPORT statement routine.....	BFE4
RESTORE statement routine.....	BAE6
RETURN statement routine.....	B8B6
RIGHT\$ function routine.....	AFEE
RND function routine.....	AF49
ROM Entry point.....	8000
ROM header.....	8005
RUN statement routine.....	BD11
Real addition.....	9C8B
Real comparison.....	9A50
Real multiplication.....	9D20
Real subtraction.....	9CE1
Remove FAC#1 from stack.....	BD7E
Remove a string from stack.....	BDDC
SAVE command routine.....	BEF3
SGN function routine.....	AB88
SIN function routine.....	A998
SOUND statement routine.....	B44C
SPC routine.....	8E58
SQR function routine.....	A7B4
STOP statement routine.....	8AD0
STR\$ function routine.....	B094
STRING\$ function routine.....	B0C2
Search for FN/PROC in catalogue.....	945B
Search for a program line.....	B99A
Search for line in program.....	9970
Search for name in catalogue.....	9469
Search for next DATA item.....	BB50
Search program for a PROC/FN.....	B112
Series evaluator.....	A897
Set (&2A)=(&2A)*(&3F).....	9236
Set (&4B)=&46C.....	A7F5
Set (&4B)=&471.....	A7E9

Set (&4B)=&476	A7ED
Set FAC#1=(&4B)+FAC#1	A500
Set FAC#1=(&4B)-FAC#1	A4FD
Set FAC#1=(&4B)/FAC#1	A6AD
Set FAC#1=0	A686
Set FAC#1=1	A699
Set FAC#1=1/FAC#1	A6A5
Set FAC#1=A	A2ED
Set FAC#1=FAC#1*(&4B)	A606
Set FAC#1=FAC#1*(&4B);chk ov/flow	A656
Set FAC#1=FAC#1*10	A1F4
Set FAC#1=FAC#1*FAC#2	A613
Set FAC#1=FAC#1+FAC#2	A50B
Set FAC#1=FAC#1-(&4B)	A4D0
Set FAC#1=FAC#1-INT(FAC#1)	A486
Set FAC#1=FAC#1/(&4B)	A6E7
Set FAC#1=FAC#1/FAC#2	A6F1
Set FAC#1=FAC#1^A	AB12
Set FAC#1=FAC#2	A4DC
Set FAC#1=IAC	A2BE
Set FAC#2=0	A453
Set FAC#2=FAC#1	A21E
Set FAC#2=FAC#1*2	A23F
Set IAC=INT(FAC#1)	A3E4
Set MAN#1=MAN#1*10	A197
Set MAN#1=MAN#1+A	A2A4
Set MAN#1=MAN#1+MAN#2	A178
Set MAN#1=MAN#2	A4E8
String addition	9C15
String assignment	8C1E
String comparison	9AE7
String concatenation	9C15
Swap FAC#1 and (&4B)	A4D6
TAB(X)	8E24
TAB(X) routine	8E40
TAN function routine	A6BE
TIME function routine	AEB4
TIME statement routine	92C9
TOP function routine	AEDC
TRACE statement routine	9295
TRUE function routine	ACC4
Token table	8071
Tokenise a line of text	8951
Tokenise line numbers	88DB
UNTIL statement routine	BBB1

USR function routine.....	ABD2
Unpack FAC#1 from (&4B).....	A3B2
Unpack FAC#2 from (&4B).....	A34E
Unstack a parameter.....	8CC1
VAL function routine.....	AC2F
VDU statement routine.....	942F
VPOS function routine.....	AB76
WIDTH statement routine.....	B4A0
Warm start.....	8AF3
Zero FAC#1.....	A686
Zero FAC#2.....	A453
^ operator routine.....	9E35

NUMERIC ROM INDEX

In numerical order, the same routines are:

8000:	ROM Entry point
8005:	ROM header
8023:	Initialisation of language
8071:	Keyword table
836D:	Keyword action addresses (LSB)
83E0:	Keyword action addresses (MSB)
8451:	Assembler mnemonics table (LSB)
848B:	Assembler mnemonics table (MSB)
84C5:	Assembler base op-codes
84FD:	Assembler exit
8504:	Assembler entry
85A5:	Assembler label definition
85BA:	Assemble a single instruction
8821:	Evaluate integer expression
883A:	EQUX in assembler
887C:	Delete bytes in buffer
889C:	Convert a line number to binary
88DB:	Line numbers; tokenise
88F3:	Line number encryption
8926:	Check if A is alphanumeric
8951:	Tokenise a line of text
8A8C:	Get character from PTR#2
8A97:	Get character from PTR#1
8AAE:	Check for a comma at PTR#2
8AB6:	OLD command routine
8AC8:	END statement routine
8AD0:	STOP statement routine
8ADA:	NEW command routine

8ADD: Cold start
 8AF3: Warm start
 8B47: = statement routine
 8B73: * statement routine
 8B7D: DATA statement routine
 8BE4: LET statement routine
 8C1E: String assignment
 8CC1: Unstack a parameter
 8D2B: PRINT# statement routine
 8D9A: PRINT statement routine
 8E24: TAB(X)
 8E40: TAB(X) routine
 8E58: SPC routine
 8EBD: CLG statement routine
 8EC4: CLS statement routine
 8ED2: CALL statement routine
 8F31: DELETE command routine
 8FA3: RENUMBER command routine
 90AC: AUTO command routine
 912F: DIM statement routine
 9222: Increment the IAC
 9236: Set (&2A)=(&2A)*(&3F)
 925D: HIMEM statement routine
 926F: LOMEM statement routine
 9283: PAGE statement routine
 928D: CLEAR statement routine
 9295: TRACE statement routine
 92C9: TIME statement routine
 9304: PROC statement routine
 9323: LOCAL statement routine
 9356: ENDPROC statement routine
 937A: GCOL statement routine
 938E: COLOUR statement routine
 939A: MODE statement routine
 93E4: MOVE statement routine
 93E8: DRAW statement routine
 93F1: PLOT statement routine
 942F: VDU statement routine
 945B: Search for FN/PROC in catalogue
 9469: Search for name in catalogue
 94ED: Create a PROC/FN catalogue entry
 94FC: Create a catalogue entry
 9531: Clear a new variable
 9559: Get an array/PROC/FN name
 95A5: ! unary operator

95A7: ? unary operator
95B0: \$ unary operator
95DD: Get a variable name
967B: ? binary operator
967F: ! binary operator
97EB: Line number decoding
9857: Check for end of statement
9880: Move to next statement
98C2: IF statement routine
991F: Print the IAC
9970: Search for line in program
99BE: Integer division
9A50: Floating point comparison
9A9E: Compare two items
9AA2: Integer comparison
9AE7: String comparison
9B29: Evaluate expression
9B3A: OR operator routine
9B55: EOR operator routine
9B72: LEVEL 2 of expression evaluator
9B7A: AND operator routine
9B9C: LEVEL 3 of expression evaluator
9BAE: = operator routine
9BC0: < operator routine
9BD4: < = operator routine
9BDF: < > operator routine
9BE8: > operator routine
9BFA: > = operator routine
9C15: String concatenation
9C42: LEVEL 4 of expression evaluator
9C4E: + operator routine
9C53: Integer addition
9C8B: Floating point addition
9CB5: - operator routine
9CBA: Integer subtraction
9CE1: Floating point subtraction
9D20: Floating point multiplication
9D3C: * operator routine
9D41: Integer multiplication
9DD1: LEVEL 5 of expression evaluator
9DE5: / operator routine
9E01: MOD operator routine
9E0A: DIV operator routine
9E20: LEVEL 6 of expression evaluator
9E35: ^ operator routine

9E90: Print number in hexadecimal
 9EDF: Print number in decimal
 A07B: Convert string to a number
 A178: Set $\text{MAN}\#1 = \text{MAN}\#1 + \text{MAN}\#2$
 A197: Set $\text{MAN}\#1 = \text{MAN}\#1 * 10$
 A1DA: Check $\text{FAC}\#1$
 A1F4: Set $\text{FAC}\#1 = \text{FAC}\#1 * 10$
 A21E: Set $\text{FAC}\#2 = \text{FAC}\#1$
 A23F: Set $\text{FAC}\#2 = \text{FAC}\#1 * 2$
 A2A4: Set $\text{MAN}\#1 = \text{MAN}\#1 + A$
 A2BE: Set $\text{FAC}\#1 = \text{IAC}$
 A2ED: Set $\text{FAC}\#1 = A$
 A303: Normalise $\text{FAC}\#1$
 A34E: Unpack $\text{FAC}\#2$ from (&4B)
 A37D: Pack $\text{FAC}\#1$ to &471 onwards
 A381: Pack $\text{FAC}\#1$ to &476 onwards
 A385: Pack $\text{FAC}\#1$ to &46C onwards
 A38D: Pack $\text{FAC}\#1$ to (&4B)
 A3B2: Unpack $\text{FAC}\#1$ from (&4B)
 A3E4: Set $\text{IAC} = \text{INT}(\text{FAC}\#1)$
 A3FE: Fix $\text{FAC}\#1$
 A453: Zero $\text{FAC}\#2$
 A486: Set $\text{FAC}\#1 = \text{FAC}\#1 - \text{INT}(\text{FAC}\#1)$
 A4B6: Increment $\text{MAN}\#1$
 A4C7: Decrement $\text{MAN}\#1$
 A4D0: Set $\text{FAC}\#1 = \text{FAC}\#1 - (\&4B)$
 A4D6: Exchange $\text{FAC}\#1$ and (&4B)
 A4DC: Set $\text{FAC}\#1 = \text{FAC}\#2$
 A4E8: Set $\text{MAN}\#1 = \text{MAN}\#2$
 A4FD: Set $\text{FAC}\#1 = (\&4B) - \text{FAC}\#1$
 A500: Set $\text{FAC}\#1 = (\&4B) + \text{FAC}\#1$
 A50B: Set $\text{FAC}\#1 = \text{FAC}\#1 + \text{FAC}\#2$
 A606: Set $\text{FAC}\#1 = \text{FAC}\#1 * (\&4B)$
 A613: Set $\text{FAC}\#1 = \text{FAC}\#1 * \text{FAC}\#2$
 A656: Set $\text{FAC}\#1 = \text{FAC}\#1 * (\&4B)$;chk ov/flow
 A686: Set $\text{FAC}\#1 = 0$
 A699: Set $\text{FAC}\#1 = 1$
 A6A5: Set $\text{FAC}\#1 = 1 / \text{FAC}\#1$
 A6AD: Set $\text{FAC}\#1 = (\&4B) / \text{FAC}\#1$
 A6BE: TAN function routine
 A6E7: Set $\text{FAC}\#1 = \text{FAC}\#1 / (\&4B)$
 A6F1: Set $\text{FAC}\#1 = \text{FAC}\#1 / \text{FAC}\#2$
 A7B4: SQR function routine
 A7E9: Set $(\&4B) = \&471$
 A7ED: Set $(\&4B) = \&476$

A7F5: Set (&4B)= &46C
 A7FE: LN function routine
 A897: Series evaluator
 A8D4: ACS function routine
 A8DA: ASN function routine
 A907: ATN function routine
 A98D: COS function routine
 A998: SIN function routine
 AA91: EXP function routine
 AB12: Set $FAC \# 1 = FAC \# 1^A$
 AB33: ADVAL function routine
 AB41: POINT function routine
 AB6D: POS function routine
 AB76: VPOS function routine
 AB88: SGN function routine
 ABA8: LOG function routine
 ABB1: RAD function routine
 ABC2: DEG function routine
 ABCB: PI function routine
 ABD2: USR function routine
 ABE9: EVAL function routine
 AC2F: VAL function routine
 AC78: INT function routine
 AC9E: ASC function routine
 ACAD: INKEY function routine
 ACB8: EOF function routine
 ACC4: TRUE function routine
 ACD1: NOT function routine
 ACE2: INSTR function routine
 AD6A: ABS function routine
 AD8C: - unary operator
 ADEC: Evaluate operand
 AE6D: Decode hexadecimal number
 AEB4: TIME function routine
 AEC0: PAGE function routine
 AECA: FALSE function routine
 AED1: LEN function routine
 AEDC: TOT function routine
 AEF7: COUNT function routine
 AEFC: LOMEM function routine
 AF03: HIMEM function routine
 AF49: RND function routine
 AF9F: ERL function routine
 AFA6: ERR function routine
 AFB9: GET function routine

AFBF: GET\$ function routine
 AFCC: LEFT\$ function routine
 AFEE: RIGHT\$ function routine
 B026: INKEY\$ function routine
 B039: MID\$ function routine
 B094: STR\$ function routine
 B0C2: STRING\$ function routine
 B112: Search program for a PROC/FN
 B195: FN function routine
 B32C: Get the value of a variable
 B3BD: CHR\$ function routine
 B3C5: Find ERL
 B402: BRK handling routine
 B433: BASIC error handler
 B44C: SOUND statement routine
 B472: ENVELOPE statement routine
 B4A0: WIDTH statement routine
 B4B1: Assign value to numeric variable
 B50E: Print a character
 B545: Print A in hexadecimal
 B58A: LISTO command routine
 B59C: LIST command routine
 B695: NEXT statement routine
 B7C4: FOR statement routine
 B888: GOSUB statement routine
 B8B6: RETURN statement routine
 B8CC: GOTO statement routine
 B8E4: ON ERROR OFF statement routine
 B8F2: ON ERROR statement routine
 B915: ON statement routine
 B99A: Search for a program line
 B9CF: INPUT# statement routine
 BA44: INPUT statement routine
 BAE6: RESTORE statement routine
 BB1F: READ statement routine
 BB50: Search for next DATA item
 BBB1: UNTIL statement routine
 BBE4: REPEAT statement routine
 BC2D: Delete a line from program
 BC8D: Insert a line into program
 BD11: RUN statement routine
 BD20: Clear variables
 BD3A: Clear stacks
 BD51: Push FAC#1 onto stack
 BD7E: Remove FAC#1 from stack

BD90: Push anything onto stack
 BD94: Push the IAC onto stack
 BDB2: Push a string onto stack
 BDCB: Pull a string from stack
 BDDC: Remove a string from stack
 BDEA: Pull the IAC from stack
 BDFF: Discard the IAC from stack
 BE62: Load a new program
 BE6F: Check for a 'Bad program'
 BEC2: OSCLI statement routine
 BEF3: SAVE command routine
 BF24: LOAD command routine
 BF2A: CHAIN statement routine
 BF30: PTR statement routine
 BF46: EXT function routine
 BF47: PTR function routine
 BF58: BPUT statement routine
 BF6F: BGET statement routine
 BF78: OPENIN function routine
 BF7C: OPENOUT function routine
 BF80: OPENUP function routine
 BF99: CLOSE statement routine
 BFA9: Get file handle
 BFCF: Print a string
 BFE4: REPORT statement routine

Here is a bastard BASIC version of the entire ROM. The level of detail in the program is not very great, but it gives a good overview of the ROM.

- 1 Initialise
- 2 Carry out a NEW
- 3 Clear the BASIC stacks
- 4 Input a line of text with '>' as the prompt to the keyboard buffer
- 5 Tokenise the line. In this stage, all the keywords are replaced with a single byte
- 6 If the line started with a line number, insert the line into the program and return to step 3
- 7 Execute the statements/commands in the line

It also helps to give an overview of the action of procedures, functions and local variables. I'll describe the operation of procedures, but functions work in a very similar manner.

When the PROC statement is encountered, the entire machine stack is pushed onto the BASIC stack. This is done to prevent stack overflow on the 6502s rather small stack.

The machine stack is then cleared and the stack pointer set to &FF. The token for 'PROC' is then placed on the machine stack, so that 'ENDPROC' will know whether a procedure or a function is executing.

The address of the procedure is then discovered, either by looking in the variable catalogue if the procedure has been called before, or by searching through the text of the program.

The current value of PTR #1 is then saved on the machine stack, for use on exit. If the procedure name in the DEF PROC statement is followed by a bracket, the computer deals with any parameters present. We will assume none are present for the moment. Zero is then pushed onto the machine stack to indicate that no parameters are present.

PTR #2 is then saved on the machine stack. It points to the address control should return to on exit from the procedure. The statements comprising the procedure are then executed. When the ENDPROC statement is encountered, a check is made to ensure a procedure is being executed, then an RTS instruction is executed to pass control back to the procedure handling routine.

The return address is then removed from the stack and placed in PTR #2. The zero byte is then removed from the stack, followed by the original value of PTR #2.

If parameters are present in the definition, each parameter is scanned in the DEF PROC statement. As each is scanned, the type and address of the parameter is saved on the machine stack and the type, address and value are stored on the BASIC stack. The calling parameters are then scanned. Each parameter is evaluated, then pushed on the BASIC stack as a type and a value. When this process is complete, a check is made to ensure the correct number of parameters were supplied.

Then, for each parameter, the type of the calling parameter is pulled from the BASIC stack, together with the type and address of the defined parameter from the machine stack. If the types are compatible, the defined parameter is set equal to the calling parameter. Finally, the number of parameters is saved on the machine stack instead of zero.

When the procedure ends, the routine recalls the original value of each defined parameter, using the information on the BASIC stack. When LOCAL variables are defined, they are zeroed after their original type, address and value is saved on the BASIC stack. The number of parameters on the machine stack is then incremented. This makes the local variable be restored at ENDPROC just like a normal parameter.

The only other area of difficulty in the ROM is the expression evaluator. The method used is not easy to explain, since it relies on the inherent nature of subroutine calls so much. Comparing the ROM code with the code used in SLUG should make the method clear.

Finally, I have used the terminology (&XXX) to indicate the 16 bit value stored in XXX and XXX + 1. This clarifies discussions of pointers and other 16 bit values.

The BASIC ROM listing

The first part of the ROM checks the contents of the accumulator. If it contains 1, the interpreter is executed, otherwise control is immediately returned

8000:cmp #&1

8002:beq &8023

8004:rts

This byte is a dummy to allow for the lack of a service entry to the BASIC ROM

8005:EA-j

This is the ROM type. &60 indicates that BASIC has no service entry, has a language entry and that a second processor relocation address is provided

8006:60-

This byte gives the address of the BASIC copyright message. In this case, the copyright message starts at &800E

8007:0E-

This byte is meant to be the version number of the ROM. It seems optimistic to label BASIC II as 'version 1'

8008:01-

The title of the ROM is stored from this byte to the copyright offset pointer. Thus, the title in this case is 'BASIC'. This title is printed when a language is entered, as in '*BASIC' or '*WORDWISE'

8009:42-B

800A:41-A

800B:53-S

800C:49-I

800D:43-C

The ROM copyright message is stored from the copyright offset pointer. It is terminated by a zero byte. The error message vector points to this address when the ROM is entered, which is why typing 'REPORT' immediately after pressing the break key or typing '*BASIC' gives the message '(C)1982 Acorn'

800E:00-

800F:28-(

8010:43-C

8011:29-)

8012:31-1

8013:39-9

8014:38-8

8015:32-2

8016:20-
 8017:41-A
 8018:63-c
 8019:6F-o
 801A:72-r
 801B:6E-n
 801C:0A-
 801D:0D-
 801E:00-

These four bytes contain the address at which the ROM is designed to be executed. For language ROMs designed to be used in the input/output processor (rather than in a second processor), this address is &00008000.

801F:00-
 8020:80-
 8021:00-
 8022:00-

Language initialisation

This section of the ROM carries out various initialisation. It is only executed when BASIC is started after '*BASIC' or 'break', and not when CLEAR or NEW are executed

Get HIMEM using OSBYTE &84

8023:lda #&84
 8025:jsr &FFF4

Save HIMEM in the relevant page zero locations

8028:stx &6
 802A:sty &7

Get PAGE using OSBYTE &83

802C:lda #&83
 802E:jsr &FFF4

Save PAGE in page zero

8031:sty &18

Set LISTO to 0

8033:ldx #&0
 8035:stx &1F

Zero the top two bytes of @%

8037:stx &402
 803A:stx &403

Set WIDTH to 255

803D:dex
 803E:stx &23

Set the print field width in @% to 10

8040:ldx #&A
 8042:stx &400

Set the number of digits printed before using E mode to 9

8045:dex**8046:stx &401**

Check if the contents of the RND store are valid

8049:lda #&1**804B:and &11****804D:ora &D****804F:ora &E****8051:ora &F****8053:ora &10**

Skip the bit of code concerned with setting the initial RND value if a valid value was found

8055:bne &8063

Set the first three bytes of the RND store to &575241. This is the RND seed

8057:lda #&41**8059:sta &D****805B:lda #&52****805D:sta &E****805F:lda #&57****8061:sta &F**

Set the BRK handler address to &B402. Thus, whenever an error occurs, control will pass to &B402

8063:lda #&2**8065:sta &202****8068:lda #&B4****806A:sta &203**

Enable interrupts. Language ROMs are always entered with interrupts disabled

806D:cli

Jump into the rest of the ROM

806E:jmp &8ADD**Keyword table**

This table contains the name of each BASIC keyword, together with two pieces of information about each keyword. The format used is:

Name of keyword

Token value

Tokenising byte

The keyword name is stored in normal ASCII format. It is terminated by the token for the keyword, which is always greater than 127. The token value is the single byte used to represent the keyword in programs

The tokenising byte contains information about the way in which each keyword should be tokenised. Each bit has the following attribute when set (the appendix lists the keywords which have each attribute)

Bit 0 - Only tokenise this keyword if the following character is not alphanumeric. This means that it is legal to have variable names that start with some keywords (those which

have this attribute bit set). This feature has been implemented because it is impossible for keywords like FALSE to be followed by anything other than a symbol such as '+' or ':'. If an alphanumeric character is encountered, the keyword is being used improperly. This can only happen legally if it is being used at the start of a variable name. The computer assumes that you were accessing a variable, and refuses to tokenise the keyword. The worst side effect of this is that a line like 'IFT%=TIMETHENPROChello' does not work, since the computer will assume you are trying to refer to a variable named 'TIMETHENPROChello'

Bit 1 - After this keyword has been encountered, stop tokenising line numbers and indicate that the start of the statement has been passed

Bit 2 - After this keyword has been encountered, go back into 'start of statement' mode

Bit 3 - Don't try to tokenise any alphanumeric characters following this token. This stops the letters following PROC and FN being tokenised. Without this feature, one would be unable to write 'DEF PROCPRINT'

Bit 4 - Allow line numbers to be tokenised after a keyword of this type

Bit 5 - Don't tokenise the rest of the characters on this line. This bit is used to stop the contents of REM and DATA statements being tokenised

Bit 6 - Add &40 to the token value if the keyword appears at the start of a statement. This is how the computer copes with the two possible tokens for words like HIMEM and PAGE (see the User Guide p.483/4)

Bit 7 - Not used

The uses of these bits will become clearer when the tokenising routine is studied

Notice that only those functions that take more than one argument include the opening bracket in the token. This is because brackets are optional with functions like 'SIN', but obligatory with 'MID\$'

8071:41-A

8072:4E-N

8073:44-D

8074:80-

8075:00-

8076:41-A

8077:42-B

8078:53-S

8079:94-

807A:00-

807B:41-A

807C:43-C

807D:53-S

807E:95-

807F:00-

8080:41-A

8081:44-D

8082:56-V

8083:41-A

8084:4C-L

8085:96-

8086:00-

8087:41-A

8088:53-S

8089:43-C

808A:97-

808B:00-

808C:41-A

808D:53-S

808E:4E-N

808F:98-

8090:00-

8091:41-A

8092:54-T

8093:4E-N

8094:99-

8095:00-

8096:41-A

8097:55-U

8098:54-T

8099:4F-O

809A:C6-F

809B:10-

809C:42-B

809D:47-G

809E:45-E

809F:54-T

80A0:9A-

80A1:01-

80A2:42-B

80A3:50-P

80A4:55-U

80A5:54-T

80A6:D5-U

80A7:03-

80A8:43-C
 80A9:4F-O
 80AA:4C-L
 80AB:4F-O
 80AC:55-U
 80AD:52-R
 80AE:FB-{
 80AF:02-

80B0:43-C
 80B1:41-A
 80B2:4C-L
 80B3:4C-L
 80B4:D6-V
 80B5:02-

80B6:43-C
 80B7:48-H
 80B8:41-A
 80B9:49-I
 80BA:4E-N
 80BB:D7-W
 80BC:02-

80BD:43-C
 80BE:48-H
 80BF:52-R
 80C0:24-\$
 80C1:BD- =
 80C2:00-

80C3:43-C
 80C4:4C-L
 80C5:45-E
 80C6:41-A
 80C7:52-R
 80C8:D8-X
 80C9:01-

80CA:43-C
 80CB:4C-L
 80CC:4F-O
 80CD:53-S
 80CE:45-E
 80CF:D9-Y
 80D0:03-

80D1:43-C
80D2:4C-L
80D3:47-G
80D4:DA-Z
80D5:01-

80D6:43-C
80D7:4C-L
80D8:53-S
80D9:DB-[
80DA:01-

80DB:43-C
80DC:4F-O
80DD:53-S
80DE:9B-
80DF:00-

80E0:43-C
80E1:4F-O
80E2:55-U
80E3:4E-N
80E4:54-T
80E5:9C-
80E6:01-

80E7:44-D
80E8:41-A
80E9:54-T
80EA:41-A
80EB:DC-/
80EC:20-

80ED:44-D
80EE:45-E
80EF:47-G
80F0:9D-
80F1:00-

80F2:44-D
80F3:45-E
80F4:46-F
80F5:DD-]
80F6:00-

80F7:44-D
80F8:45-E
80F9:4C-L
80FA:45-E

80FB:54-T
 80FC:45-E
 80FD:C7-G
 80FE:10-

80FF:44-D
 8100:49-I
 8101:56-V
 8102:81-
 8103:00-

8104:44-D
 8105:49-I
 8106:4D-M
 8107:DE-^
 8108:02-

8109:44-D
 810A:52-R
 810B:41-A
 810C:57-W
 810D:DF-__
 810E:02-

810F:45-E
 8110:4E-N
 8111:44-D
 8112:50-P
 8113:52-R
 8114:4F-O
 8115:43-C
 8116:E1-a
 8117:01-

8118:45-E
 8119:4E-N
 811A:44-D
 811B:E0-‘

811C:01-
 811D:45-E
 811E:4E-N
 811F:56-V
 8120:45-E
 8121:4C-L
 8122:4F-O
 8123:50-P
 8124:45-E

8125:E2-b

8126:02-

8127:45-E

8128:4C-L

8129:53-S

812A:45-E

812B:8B-

812C:14-

812D:45-E

812E:56-V

812F:41-A

8130:4C-L

8131:A0-

8132:00-

8133:45-E

8134:52-R

8135:4C-L

8136:9E-

8137:01-

8138:45-E

8139:52-R

813A:52-R

813B:4F-O

813C:52-R

813D:85-

813E:04-

813F:45-E

8140:4F-O

8141:46-F

8142:C5-E

8143:01-

8144:45-E

8145:4F-O

8146:52-R

8147:82-

8148:00-

8149:45-E

814A:52-R

814B:52-R

814C:9F-

814D:01-

814E:45-E

814F:58-X

8150:50-P

8151:A1-!

8152:00-

8153:45-E

8154:58-X

8155:54-T

8156:A2-"

8157:01-

8158:46-F

8159:4F-O

815A:52-R

815B:E3-c

815C:02-

815D:46-F

815E:41-A

815F:4C-L

8160:53-S

8161:45-E

8162:A3-#

8163:01-

8164:46-F

8165:4E-N

8166:A4-\$

8167:08-

8168:47-G

8169:4F-O

816A:54-T

816B:4F-O

816C:E5-e

816D:12-

816E:47-G

816F:45-E

8170:54-T

8171:24-\$

8172:BE->

8173:00-

8174:47-G

8175:45-E

8176:54-T

8177:A5-%

8178:00-

8179:47-G

817A:4F-O

817B:53-S

817C:55-U

817D:42-B

817E:E4-d

817F:12-

8180:47-G

8181:43-C

8182:4F-O

8183:4C-L

8184:E6-f

8185:02-

8186:48-H

8187:49-I

8188:4D-M

8189:45-E

818A:4D-M

818B:93-

818C:43-C

818D:49-I

818E:4E-N

818F:50-P

8190:55-U

8191:54-T

8192:E8-h

8193:02-

8194:49-I

8195:46-F

8196:E7-g

8197:02-

8198:49-I

8199:4E-N

819A:4B-K

819B:45-E

819C:59-Y

819D:24-\$

819E:BF-?

819F:00-

81A0:49-I
81A1:4E-N
81A2:4B-K
81A3:45-E
81A4:59-Y
81A5:A6-&
81A6:00-

81A7:49-I
81A8:4E-N
81A9:54-T
81AA:A8-(
81AB:00-

81AC:49-I
81AD:4E-N
81AE:53-S
81AF:54-T
81B0:52-R
81B1:28-(
81B2:A7-'
81B3:00-

81B4:4C-L
81B5:49-I
81B6:53-S
81B7:54-T
81B8:C9-I
81B9:10-

81BA:4C-L
81BB:49-I
81BC:4E-N
81BD:45-E
81BE:86-
81BF:00-

81C0:4C-L
81C1:4F-O
81C2:41-A
81C3:44-D
81C4:C8-H
81C5:02-

81C6:4C-L
81C7:4F-O
81C8:4D-M
81C9:45-E

81CA:4D-M

81CB:92-

81CC:43-C

81CD:4C-L

81CE:4F-O

81CF:43-C

81D0:41-A

81D1:4C-L

81D2:EA-j

81D3:02-

81D4:4C-L

81D5:45-E

81D6:46-F

81D7:54-T

81D8:24-\$

81D9:28-(

81DA:C0-@

81DB:00-

81DC:4C-L

81DD:45-E

81DE:4E-N

81DF:A9-)

81E0:00-

81E1:4C-L

81E2:45-E

81E3:54-T

81E4:E9-i

81E5:04-

81E6:4C-L

81E7:4F-O

81E8:47-G

81E9:AB- +

81EA:00-

81EB:4C-L

81EC:4E-N

81ED:AA-*

81EE:00-

81EF:4D-M

81F0:49-I

81F1:44-D

81F2:24-\$

81F3:28-(

81F4:C1-A
81F5:00-

81F6:4D-M
81F7:4F-O
81F8:44-D
81F9:45-E
81FA:EB-k
81FB:02-

81FC:4D-M
81FD:4F-O
81FE:44-D
81FF:83-
8200:00-

8201:4D-M
8202:4F-O
8203:56-V
8204:45-E
8205:EC-1
8206:02-

8207:4E-N
8208:45-E
8209:58-X
820A:54-T
820B:ED-m
820C:02-

820D:4E-N
820E:45-E
820F:57-W
8210:CA-J
8211:01-

8212:4E-N
8213:4F-O
8214:54-T
8215:AC-,
8216:00-

8217:4F-O
8218:4C-L
8219:44-D
821A:CB-K
821B:01-

821C:4F-O
821D:4E-N
821E:EE-n
821F:02-

8220:4F-O
8221:46-F
8222:46-F
8223:87-
8224:00-

8225:4F-O
8226:52-R
8227:84-
8228:00-

8229:4F-O
822A:50-P
822B:45-E
822C:4E-N
822D:49-I
822E:4E-N
822F:8E-
8230:00-

8231:4F-O
8232:50-P
8233:45-E
8234:4E-N
8235:4F-O
8236:55-U
8237:54-T
8238:AE-.
8239:00-

823A:4F-O
823B:50-P
823C:45-E
823D:4E-N
823E:55-U
823F:50-P
8240:AD--
8241:00-

8242:4F-O
8243:53-S
8244:43-C
8245:4C-L

8246:49-I
8247:FF-
8248:02-

8249:50-P
824A:52-R
824B:49-I
824C:4E-N
824D:54-T
824E:F1-q
824F:02-

8250:50-P
8251:41-A
8252:47-G
8253:45-E
8254:90-
8255:43-C

8256:50-P
8257:54-T
8258:52-R
8259:8F-
825A:43-C

825B:50-P
825C:49-I
825D:AF-/
825E:01-

825F:50-P
8260:4C-L
8261:4F-O
8262:54-T
8263:F0-p
8264:02-

8265:50-P
8266:4F-O
8267:49-I
8268:4E-N
8269:54-T
826A:28-(
826B:B0-0
826C:00-

826D:50-P
826E:52-R
826F:4F-O

8270:43-C

8271:F2-r

8272:0A-

8273:50-P

8274:4F-O

8275:53-S

8276:B1-1

8277:01-

8278:52-R

8279:45-E

827A:54-T

827B:55-U

827C:52-R

827D:4E-N

827E:F8-x

827F:01-

8280:52-R

8281:45-E

8282:50-P

8283:45-E

8284:41-A

8285:54-T

8286:F5-u

8287:00-

8288:52-R

8289:45-E

828A:50-P

828B:4F-O

828C:52-R

828D:54-T

828E:F6-v

828F:01-

8290:52-R

8291:45-E

8292:41-A

8293:44-D

8294:F3-s

8295:02-

8296:52-R

8297:45-E

8298:4D-M

8299:F4-t

829A:20-

829B:52-R

829C:55-U

829D:4E-N

829E:F9-y

829F:01-

82A0:52-R

82A1:41-A

82A2:44-D

82A3:B2-2

82A4:00-

82A5:52-R

82A6:45-E

82A7:53-S

82A8:54-T

82A9:4F-O

82AA:52-R

82AB:45-E

82AC:F7-w

82AD:12-

82AE:52-R

82AF:49-I

82B0:47-G

82B1:48-H

82B2:54-T

82B3:24-\$

82B4:28-(

82B5:C2-B

82B6:00-

82B7:52-R

82B8:4E-N

82B9:44-D

82BA:B3-3

82BB:01-

82BC:52-R

82BD:45-E

82BE:4E-N

82BF:55-U

82C0:4D-M

82C1:42-B

82C2:45-E

82C3:52-R

82C4:CC-L

82C5:10-

82C6:53-S

82C7:54-T

82C8:45-E

82C9:50-P

82CA:88-

82CB:00-

82CC:53-S

82CD:41-A

82CE:56-V

82CF:45-E

82D0:CD-M

82D1:02-

82D2:53-S

82D3:47-G

82D4:4E-N

82D5:B4-4

82D6:00-

82D7:53-S

82D8:49-I

82D9:4E-N

82DA:B5-5

82DB:00-

82DC:53-S

82DD:51-Q

82DE:52-R

82DF:B6-6

82E0:00-

82E1:53-S

82E2:50-P

82E3:43-C

82E4:89-

82E5:00-

82E6:53-S

82E7:54-T

82E8:52-R

82E9:24-\$

82EA:C3-C

82EB:00-

82EC:53-S
82ED:54-T
82EE:52-R
82EF:49-I
82F0:4E-N
82F1:47-G
82F2:24-\$
82F3:28-(
82F4:C4-D
82F5:00-

82F6:53-S
82F7:4F-O
82F8:55-U
82F9:4E-N
82FA:44-D
82FB:D4-T
82FC:02-

82FD:53-S
82FE:54-T
82FF:4F-O
8300:50-P
8301:FA-z
8302:01-

8303:54-T
8304:41-A
8305:4E-N
8306:B7-7
8307:00-

8308:54-T
8309:48-H
830A:45-E
830B:4E-N
830C:8C-
830D:14-

830E:54-T
830F:4F-O
8310:B8-8
8311:00-

8312:54-T
8313:41-A
8314:42-B
8315:28-(

8316:8A-

8317:00-

8318:54-T

8319:52-R

831A:41-A

831B:43-C

831C:45-E

831D:FC-|

831E:12-

831F:54-T

8320:49-I

8321:4D-M

8322:45-E

8323:91-

8324:43-C

8325:54-T

8326:52-R

8327:55-U

8328:45-E

8329:B9-9

832A:01-

832B:55-U

832C:4E-N

832D:54-T

832E:49-I

832F:4C-L

8330:FD-}

8331:02-

8332:55-U

8333:53-S

8334:52-R

8335:BA-:

8336:00-

8337:56-V

8338:44-D

8339:55-U

833A:EF-o

833B:02-

833C:56-V

833D:41-A

833E:4C-L

833F:BB-;
8340:00-

8341:56-V
8342:50-P
8343:4F-O
8344:53-S
8345:BC-<
8346:01-

8347:57-W
8348:49-I
8349:44-D
834A:54-T
834B:48-H
834C:FE-__
834D:02-

All the keywords are stored in more or less alphabetical order, except where conflicts could arise - for example, the computer must check for 'ENDPROC' before it checks for 'END', otherwise the first three letters of 'ENDPROC' will be interpreted as the quite different keyword 'END'. However, these last five keywords are out of order. This is because they are the versions of the pseudo-variables that are used on the left of the equals sign. They are not needed by the routine which does the encoding into tokens, since the keywords already exist earlier in the table, but they are needed by the routine which converts tokens into alphanumeric sequences

834E:50-P
834F:41-A
8350:47-G
8351:45-E
8352:D0-P
8353:00-

8354:50-P
8355:54-T
8356:52-R
8357:CF-O
8358:00-

8359:54-T
835A:49-I
835B:4D-M
835C:45-E
835D:D1-Q
835E:00-

835F:4C-L
8360:4F-O

8361:4D-M**8362:45-E****8363:4D-M****8364:D2-R****8365:00-****8366:48-H****8367:49-I****8368:4D-M****8369:45-E****836A:4D-M****836B:D3-S****836C:00-****LSB keyword action addresses**

This table contains the LSBs of the addresses that should be called when each keyword token is encountered. The first address in the table corresponds to OPENIN, which has a token value of &8E and an action address of &BF78

836D:78-x**836E:47-G****836F:C0-@****8370:B4-4****8371:FC-|****8372:03-****8373:6A-j****8374:D4-T****8375:33-3****8376:9E-****8377:DA-Z****8378:07-****8379:6F-o****837A:8D-****837B:F7-w****837C:C2-B****837D:9F-****837E:A6-&****837F:E9-i****8380:91-****8381:46-F****8382:CA-J****8383:95-****8384:B9-9****8385:AD--****8386:E2-b****8387:78-x**

8388:D1-Q
8389:FE-__
838A:A8-(
838B:D1-Q
838C:80-
838D:7C-|
838E:CB-K
838F:41-A
8390:6D-m
8391:B1-1
8392:49-I
8393:88-
8394:98-
8395:B4-4
8396:BE->
8397:DC-/
8398:C4-D
8399:D2-R
839A:2F-/
839B:76-v
839C:BD- =
839D:BF-?
839E:26-&
839F:CC-L
83A0:39-9
83A1:EE-n
83A2:94-
83A3:C2-B
83A4:B8-8
83A5:AC-,
83A6:31-1
83A7:24-\$
83A8:9C-
83A9:DA-Z
83AA:B6-6
83AB:A3-#
83AC:F3-s
83AD:2A-★
83AE:30-0
83AF:83-
83B0:C9-I
83B1:6F-o
83B2:5D-]
83B3:4C-L
83B4:58-X

83B5:D2-R
83B6:2A-★
83B7:8D-
83B8:99-
83B9:BD- =
83BA:C4-D
83BB:7D-}
83BC:7D-}
83BD:2F-/
83BE:E8-h
83BF:C8-H
83C0:56-V
83C1:72-r
83C2:C4-D
83C3:88-
83C4:CC-L
83C5:7A-z
83C6:C2-B
83C7:44-D
83C8:E4-d
83C9:23-#
83CA:9A-
83CB:E4-d
83CC:95-
83CD:15-
83CE:2F-/
83CF:F1-q
83D0:9A-
83D1:04-
83D2:1F-
83D3:7D-}
83D4:E4-d
83D5:E4-d
83D6:E6-f
83D7:B6-6
83D8:11-
83D9:D0-P
83DA:8E-
83DB:95-
83DC:B1-1
83DD:A0-
83DE:C2-B

MSB keyword action addresses

This table gives the MSBs of the action addresses of all the keywords. As with the previous table, the first entry corresponds to OPENIN

83DF:BF-?
 83E0:BF-?
 83E1:AE-.
 83E2:AE-.
 83E3:AE-.
 83E4:AF-/
 83E5:AD--
 83E6:A8-(
 83E7:AB- +
 83E8:AC-,
 83E9:A8-(
 83EA:A9-)
 83EB:BF-?
 83EC:A9-)
 83ED:AE-.
 83EE:AB- +
 83EF:AF-/
 83F0:AF-/
 83F1:AB- +
 83F2:AA-★
 83F3:BF-?
 83F4:AE-.
 83F5:B1-1
 83F6:AF-/
 83F7:AC-,
 83F8:AC-,
 83F9:AC-,
 83FA:AE-.
 83FB:A7-'
 83FC:AB- +
 83FD:AC-,
 83FE:BF-?
 83FF:BF-?
 8400:AB- +
 8401:AB- +
 8402:AB- +
 8403:AB- +
 8404:AF-/
 8405:AB- +
 8406:A9-)
 8407:A7-'
 8408:A6-&
 8409:AE-.
 840A:AC-,
 840B:AB- +

840C:AC-,
840D:AB- +
840E:B3-3
840F:AF-/
8410:B0-0
8411:AF-/
8412:B0-0
8413:AF-/
8414:B0-0
8415:B0-0
8416:AC-,
8417:90-
8418:8F-
8419:BF-?
841A:B5-5
841B:8A-
841C:8A-
841D:8F-
841E:BE->
841F:98-
8420:BF-?
8421:92-
8422:92-
8423:92-
8424:92-
8425:B4-4
8426:BF-?
8427:8E-
8428:BF-?
8429:92-
842A:BF-?
842B:8E-
842C:8E-
842D:8B-
842E:8B-
842F:91-
8430:93-
8431:8A-
8432:93-
8433:B4-4
8434:B7-7
8435:B8-8
8436:B8-8
8437:93-
8438:98-

8439:BA-:
 843A:8B-
 843B:93-
 843C:93-
 843D:93-
 843E:B6-6
 843F:B9-9
 8440:94-
 8441:93-
 8442:8D-
 8443:93-
 8444:BB-;
 8445:8B-
 8446:BB-;
 8447:BF-?
 8448:BA-:
 8449:B8-8
 844A:BD- =
 844B:8A-
 844C:93-
 844D:92-
 844E:BB-;
 844F:B4-4
 8450:BE->

LSB assembler mnemonics

This table, together with the next one is used to store the names of the 6502 mnemonics supported by the assembler, including OPT and EQU, the pseudo mnemonics

Each mnemonic is exactly three letters long, which might lead you to expect that three tables would be required to store the mnemonics (one for each letter). However, each mnemonic is compressed into 15 bits, which is then split into two tables (one is eight bits wide, the other is seven bits wide).

The compression method used is simple: each mnemonic contains only letters. There are 26 letters in the alphabet, so we can represent any single letter in just five bits - in other words, with a number from 0 to 31. If five bits are needed for each letter, 15 bits are required for an entire mnemonic. The compression is effected by dividing the 16 bits of the destination into four fields:

1 bit - 5 bits - 5 bits - 5 bits

Each of the five bit fields is used to contain a single letter.

The order in which the mnemonics are stored is unusual:

01 BRK
 02 CLC

THE BBC MICRO COMPENDIUM

03 CLD
04 CLI
05 CLV
06 DEX
07 DEY
08 INX
09 INY
0A NOP
0B PHA
0C PHP
0D PLA
0E PLP
0F RTI
10 RTS
11 SEC
12 SED
13 SEI
14 TAX
15 TAY
16 TSX
17 TXA
18 TXS
19 TYA
1A BCC
1B BCS
1C BEQ
1D BMI
1E BNE
1F BPL
20 BVC
21 BVS
22 AND
23 EOR
24 ORA
25 ADC
26 CMP
27 LDA
28 SBC
29 ASL
2A LSR
2B ROL
2C ROR
2D DEC
2E INC
2F CPX

30 CPY
31 BIT
32 JMP
33 JSR
34 LDX
35 LDY
36 STA
37 STX
38 STY
39 OPT
3A EQU

A fairly casual inspection of the above order will show the logic involved. The instructions are grouped together according to the addressing modes they allow.

The first of the two tables gives the LSB of the mnemonics

8451:4B-K
8452:83-
8453:84-
8454:89-
8455:96-
8456:B8-8
8457:B9-9
8458:D8-X
8459:D9-Y
845A:F0-p
845B:01-
845C:10-
845D:81-
845E:90-
845F:89-
8460:93-
8461:A3-#
8462:A4-\$
8463:A9-)
8464:38-8
8465:39-9
8466:78-x
8467:01-
8468:13-
8469:21-!
846A:63-c
846B:73-s
846C:B1-1
846D:A9-)
846E:C5-E

846F:0C-
 8470:C3-C
 8471:D3-S
 8472:C4-D
 8473:F2-r
 8474:41-A
 8475:83-
 8476:B0-0
 8477:81-
 8478:43-C
 8479:6C-1
 847A:72-r
 847B:EC-1
 847C:F2-r
 847D:A3-#
 847E:C3-C
 847F:18-
 8480:19-
 8481:34-4
 8482:B0-0
 8483:72-r
 8484:98-
 8485:99-
 8486:81-
 8487:98-
 8488:99-
 8489:14-
 848A:35-5

MSB assembler mnemonics

This table contains the upper seven bytes of the 15 bit representation of each mnemonic.

848B:0A-
 848C:0D-
 848D:0D-
 848E:0D-
 848F:0D-
 8490:10-
 8491:10-
 8492:25-%
 8493:25-%
 8494:39-9
 8495:41-A
 8496:41-A
 8497:41-A

8498:41-A
8499:4A-J
849A:4A-J
849B:4C-L
849C:4C-L
849D:4C-L
849E:50-P
849F:50-P
84A0:52-R
84A1:53-S
84A2:53-S
84A3:53-S
84A4:08-
84A5:08-
84A6:08-
84A7:09-
84A8:09-
84A9:0A-
84AA:0A-
84AB:0A-
84AC:05-
84AD:15-
84AE:3E->
84AF:04-
84B0:0D-
84B1:30-0
84B2:4C-L
84B3:06-
84B4:32-2
84B5:49-I
84B6:49-I
84B7:10-
84B8:25-%
84B9:0E-
84BA:0E-
84BB:09-
84BC:29-)
84BD:2A-★
84BE:30-0
84BF:30-0
84C0:4E-N
84C1:4E-N
84C2:4E-N
84C3:3E->
84C4:16-

Base op-codes

This table contains the base op-code for each instruction. The base op-code is used to derive the actual op-code for the particular addressing mode being used. For the first few instructions, which use inherent and relative addressing, the base op-code is the actual op-code used

BRK:**84C5:00-****CLC:****84C6:18-****CLD:****84C7:D8-X****CLI:****84C8:58-X****CLV:****84C9:B8-8****DEX:****84CA:CA-J****DEY:****84CB:88-****INX:****84CC:E8-h****INY:****84CD:C8-H****NOP:****84CE:EA-j****PHA:****84CF:48-H****PHP:****84D0:08-****PLA:****84D1:68-h****PLP:**

84D2:28-(
 RTI:
 84D3:40-@
 RTS:
 84D4:60-'
 SEC:
 84D5:38-8
 SED:
 84D6:F8-x
 SEI:
 84D7:78-x
 TAX:
 84D8:AA-★
 TAY:
 84D9:A8-(
 TSX:
 84DA:BA-:
 TXA:
 84DB:8A-
 TXS:
 84DC:9A-
 TYA:
 84DD:98-
 BCC:
 84DE:90-
 BCS:
 84DF:B0-0
 BEQ:
 84E0:F0-p
 BMI:

84E1:30-0

BNE:

84E2:D0-P

BPL:

84E3:10-

BVC:

84E4:50-P

BVS:

84E5:70-p

AND:

84E6:21-!

EOR:

84E7:41-A

ORA:

84E8:01-

ADC:

84E9:61-a

CMP:

84EA:C1-A

LDA:

84EB:A1-!

SBC:

84EC:E1-a

ASL:

84ED:06-

LSR:

84EE:46-F

ROL:

84EF:26-&

ROR:

84F0:66-f**DEC:****84F1:C6-F****INC:****84F2:E6-f****CPX:****84F3:E0-‘****CPY:****84F4:C0-@****BIT:****84F5:20-****JMP:****84F6:4C-L****JSR:****84F7:20-****LDX:****84F8:A2-"****LDY:****84F9:A0-****STA:****84FA:81-****STX:****84FB:86-****STY:****84FC:84-****Assembler exit**

Before the assembler makes a return to the normal BASIC interpreter, it places the value &FF into the location used to keep track of the current OPT value. This is done because the part of the ROM concerned with evaluating expressions will only give a 'No such variable' message if OPT signals that errors are to be flagged. In normal BASIC processing, it is necessary to make sure that all errors are flagged. If OPT is set to zero, which indicates that errors are not to be flagged, any undefined variables are reported to have the value of P%. You can try this with a line like:

P% = 12345: ? & 28 = 0: PRINT NO__SUCH__VARIABLE

Unfortunately, the variable 'NO__SUCH__VARIABLE' will not be created by the above line, so this trick is not particularly useful.

Save &FF as the current OPT value

84FD:lda #&FF

84FF:sta &28

Join up with the BASIC interpreter

8501:jmp &8BA3

Assembler entry point

Whenever a '[' is encountered where a statement might otherwise be expected, control is passed here.

Set the OPT value to the default of 3

8504:lda #&3

8506:sta &28

Get the next character

8508:jsr &8A97

Exit if it is a ']'

850B:cmp #&5D

850D:beq &84FD

Add Y to (&B), which makes PTR#1 point directly to the start of the assembly language statement

850F:jsr &986D

Decrement the offset to zero. (The routine at &986D left the offset as 1, effectively incrementing the pointer at the same time)

8512:dec &A

Assemble a single instruction

8514:jsr &85BA

Decrement PTR#1 to the last character that couldn't be assembled. This will usually be the colon or carriage return after the instruction, but it could be the comment following the instruction - '**LDA &200 get the next part**' is a legal instruction

8517:dec &A

Copy bit zero of OPT into the carry flag. This bit is used to determine whether or not a listing should be given.

8519:lda &28

851B:lsr A

Skip the code that gives a listing if one is not required

851C:bcc &857E

Get the current value of COUNT and add five to it (the carry flag will always be set before the ADC instruction is executed). This value is the column after the address in the listing. It is stored so that a line like 'PRINT "*****";[EQU "*****":]' will look reasonable when listed. If you try this line in immediate mode you will see the way instructions are lined up, even though the first line is preceded by asterisks.

851E:lda &1E**8520:adc #&4**

Store this value in a temporary location

8522:sta &3F

Get the MSB of the code pointer. This pointer is the same as P% was before the instruction was assembled

8524:lda &38

Print it in hex

8526:jsr &B545

Get the LSB of the code pointer

8529:lda &37

Print that in hex followed by a space

852B:jsr &B562

Up to 255 bytes can be generated by a single instruction, but the listing tabulates all the bytes into columns of 3 bytes each. The X register is used to keep track of how many more bytes can be printed before a new line is started. The above example shows this effect too. The value &FC is loaded since it corresponds to the two's complement value of -4. Thus, the fourth time it is incremented, the zero flag will be set, which indicates that a new line is required

852E:ldx #&FC

Get the number of bytes that were generated

8530:ldy &39

If a string was assembled, the code length is set as &FF and the real length of the string can be found in the normal place (&36). These two instructions change the code length to the length of the string if necessary

8532:bpl &8536**8534:ldy &36**

Save the code length

8536:sty &38

Skip the rest of the listing if no code was generated. This situation arises with lines that are only made up of comments

8538:beq &8556

Y is used as a pointer into the code generated as the listing is given. It is set to zero to ensure that the first byte printed is the first byte that was assembled

853A:ldy #&0

Increment the counter to see if a new line has to be started

853C:inx

Skip the new line code if a new line is not required

853D:bne &854C

Move the cursor to a new line

853F:jsr &BC25

Get the offset of the listing from the left of the screen

8542:ldx &3F

Print a space

8544:jsr &B565

Continue printing spaces until enough have been printed

8547:dex

8548:bne &8544

Set X to &FD to continue keeping count of the number of bytes on the current line. This is done because on the second loop through this code, the first INX instruction will not be executed

854A:ldx #&FD

Get the next byte of the assembled code

854C:lda (&3A),Y

Print it in hexadecimal, followed by a space

854E:jsr &B562

Point to the next byte assembled

8551:iny

Go back for more if all the bytes have not been printed

8552:dec &38

8554:bne &853C

Use X to check if the end of the line has been reached

8556:inx

Skip printing dummy spaces if the end of the line has been reached

8557:bpl &8565

Print three spaces to take the place of a non-existent byte

8559:jsr &B565

855C:jsr &B558

855F:jsr &B558

Skip backwards to continue this process

8562:jmp &8556

Zero Y, which will be used as a pointer into the source text of the instruction just assembled. Using this pointer, the source text can then be printed

8565:ldy #&0

Get the next character of the source text

8567:lda (&B),Y

Skip forwards if the character is a colon. The colon could be in a statement of the form 'LDA ASC(":")', so more tests are carried out further on

8569:cmp #&3A

856B:beq &8577

Skip farther forwards if we have reached the end of the line

856D:cmp #&D

856F:beq &857B

Print the character, changing tokens to ASCII strings

8571:jsr &B50E

Point to the next character

8574:iny

Jump backwards to deal with the new character. The BNE instruction in this case will always be executed, since Y cannot be incremented to zero without having a statement that is 256 characters long - which is impossible

8575:bne &8567

This section checks if the colon encountered does actually signal the end of the statement. It compares the current pointer with the pointer value offset when the statement had been assembled

8577:cpy &A

If Y has not yet reached this value, the end of the statement has not been reached, so the process continues

8579:bcc &8571

Move the cursor to a new line

857B:jsr &BC25

Load Y with the offset of the first character that couldn't be assembled

857E:ldy &A

Decrement Y to account for the first INY instruction in the loop that follows

8580:dey

Point to the next character

8581:iny

Retrieve the character

8582:lda (&B),Y

Skip forwards if it was a colon

8584:cmp #&3A**8586:beq &858C**

Skip backwards if it was not a carriage return. Thus, this loop has skipped any extraneous characters between the end of the statement and the colon/carriage return at the end of the statement

8588:cmp #&D**858A:bne &8581**

Check the statement/line has ended and add Y to PTR#1

858C:jsr &9859

Check if the last statement terminated with a colon

858F:dey**8590:lda (&B),Y****8592:cmp #&3A**

Skip forwards if it did

8594:beq &85A2

Otherwise, check for immediate mode by seeing if the text pointer points to the keyboard buffer

8596:lda &C**8598:cmp #&7**

Skip the next instruction if immediate mode is being used

859A:bne &859F

Jump to the cold start routine

859C:jmp &8AF6

Move PTR#1 to the start of the next statement

859F:jsr &9890

Jump back to assemble the next instruction

85A2:jmp &8508**Label definitions**

This short section of code deals with a label definition being found in place of a normal assembly language mnemonic. It falls straight into the 'Assemble single instruction' routine, which is why '.loop jmp loop' is a legal instruction.

Get the variable name, check the variable exists, and create it if it doesn't exist. After this routine, &2A,&2B contain the address of the value of the variable, while &2C contains the type of the variable (the type indication system used is the same as that used in the CALL statement - see p.215 of the User Guide)

85A5:jsr &9582

Give a 'Syntax error' if the variable was badly formed or was a string

85A8:beq &8604**85AA:bcx &8604**

Save the IAC on the BASIC stack; this saves the address and type of the variable

85AC:jsr &BD94

Place P% in IAC

85AF:jsr &AE3A

Set the type as an integer, since the accumulator will contain &40 from the routine at &AE3A

85B2:sta &27

Assign the value in IAC to the variable. The routine at &B4B4 copes with things like the variable being real, in which case the IAC has to be converted to FAC#1

85B4:jsr &B4B4

Update the offset of PTR#1 from the offset of PTR#2, since the routine used to get a variable name used PTR#2

85B7:jsr &8827**Assemble a single instruction**

This routine assembles a single instruction from PTR#1. It is also one of the most complex routines in the ROM, so it is important not to worry if you don't fully understand it on the first few readings. It is effectively a complete 6502 assembler in less than 1K

Indicate that there are three letters remaining to be found in the current mnemonic

85BA:ldx #&3

Get the next character

85BC:jsr &8A97

Zero the top of the mnemonic buffer. The 15 bit representation of the mnemonic will be stored in &3D and &3E. It will be put there using the ROL operation, thus, we have to zero the LSB of the buffer to ensure that the very top bit will be zero. This point is quite subtle, but an examination of the following code should make it clear

85BF:ldy #&0**85C1:sty &3D**

Skip assembly if a colon is encountered in place of any instruction character

85C3:cmp #&3A

85C5:beq &862B

Deal similarly with the end of the line

85C7:cmp #&D

85C9:beq &862B

And the ']' character

85CB:cmp #&5C

85CD:beq &862B

Go back to the code to deal with labels if a full stop is encountered. This shows that a statement like '.loop .A% .B% NOP' is legal

85CF:cmp #&2E

85D1:beq &85A5

Decrement the offset of PTR # 1. This is to account for the character that we retrieved at &85BC

85D3:dec &A

Load Y from the offset of PTR # 1

85D5:ldy &A

Increment the offset to the next character

85D7:inc &A

Get the first character of the mnemonic

85D9:lda (&B),Y

Skip to location &8607 if the character is a token. This is needed because three of the instructions start with tokens (ORA, EOR and AND)

85DB:bmi &8607

If the character was a space, the end of the mnemonic has been prematurely encountered, so the rest of the code to get the mnemonic is skipped

85DD:cmp #&20

85DF:beq &85F1

Use Y to indicate that there are five more bits in the character that we need to deal with. Thus, only the bottom five bits of each character in a mnemonic are dealt with. In turn, this means that we can write mnemonics in upper or lower case. More subtly, it also means that we can write instructions using any characters whose bottom five bits are correct. If you turn to the ASCII table on page 492 of the User Guide, you'll see that the bottom five bits of the character '\$' are the same as those of 'D'. This allows us to write 'A\$C' instead of 'ADC'. For example, these three instructions are common: '" 2+', '!3,' and ').'

85E1:ldy #&5

These three instructions shift the character left three times. This makes the bottom five bits of the character become the top five bits

85E3:asl A

85E4:asl A

85E5:asl A

Get the top bit of the character

85E6:asl A

Move it into the mnemonic buffer

85E7:rol &3D

85E9:rol &3E

Continue this process until all five bits have been dealt with

85EB:dey

85EC:bne &85E6

Continue until all three letters of the mnemonic have been read in

85EE:dex

85EF:bne &85D5

Make X point to the last instruction mnemonic in the list

85F1:ldx #&3A

Get the LSB of the current instruction mnemonic

85F3:lda &3D

Compare it against the current table entry

85F5:cmp &8450,X

Skip if no match is found

85F8:bne &8601

Get the current MSB table entry

85FA:ldy &848A,X

Compare it against the MSB of the current mnemonic

85FD:cpy &3E

Exit if a match is detected. In this case, X contains the number (from the table given earlier) of the instruction in question

85FF:beq &8620

Check against the next instruction in the list

8601:dex

8602:bne &85F5

Signal a 'Syntax error', since the end of the list has been reached without a match being found

8604:jmp &982A

This is the routine that deals with tokens being found in the mnemonic. It starts by loading X with the code for the 'AND' instruction

8607:ldx #&22

Check if the token encountered was 'AND'

8609:cmp #&80

Exit if it was (X will contain the correct instruction code)

860B:beq &8620

Increment X, so that it contains the instruction code for 'EOR'

860D:inx

Check the token found against 'EOR'

860E:cmp #&82

Exit if the tokens matched

8610:beq &8620

Make X contain the instruction code for 'ORA'

8612:inx

Check that the token found was 'OR'

8613:cmp #&84

Give a 'Syntax error' if it was not

8615:bne &8604

Point to the next letter following 'OR'

8617:inc &A

Increment Y, which also points to the next letter

8619:iny

Get the next letter

861A:lda (&B),Y

Give a 'Syntax error' if it was not 'A'

861C:cmp #&41

861E:bne &8604

Control passes to here when X contains a valid instruction code

The routine first extracts the base op-code for the instruction in question

8620:lda &84C4,X

Store the op-code in location &29

8623:sta &29

Y is used to indicate the length of code that has been generated. For the moment, we are assuming that all instructions are using inherent addressing, which implies that instructions have a length of one byte; hence, Y is initialised to one.

8625:ldy #&1

Branch to &8673 if the instruction code was $\geq \&1A$. This makes control pass to &8673 for all instructions that do not employ inherent addressing

8627:cpx #&1A

8629:bcs &8673

Control returns to here when an instruction has been properly assembled. The arguments of the instruction are currently in the IAC (if necessary), with the op-code being in &29. The code length is given by Y. The first thing the routine does is to get the LSB of P%

862B:lda &440

Store it in &37

862E:sta &37

Save the code length in &39

8630:sty &39

Get the current OPT value

8632:ldx &28

Compare it with 4. If OPT is greater than 4, it implies that O% is in use as the code origin

8634:cpx #&4

Get the MSB of P% and store it in &38. This now makes (&37) contain the address at which the instruction has been assembled. This data is also stored in X(MSB) and A(LSB)

8636:ldx &441

8639:stx &38

If the OPT value was less than 4, skip the next two instructions

863B:bcc &8643

Change X and A to reflect the value of O%. X and A will now be used as the code origin, since they will contain P% if OPT is less than 4 and O% otherwise

863D:lda &43C

8640:ldx &43D

Save the code origin in (&3A)

8643:sta &3A

8645:stx &3B

Copy the code length to A. This is done to make the flags reflect the status of the code length

8647:tya

Exit if no code was generated

8648:beq &8672

If the code length is not &FF, we have a true representation of the length of the code, so skip the next instructions

864A:bpl &8650

Otherwise, get the code length from the byte indicating the length of the string in the string buffer - this occurs when EQUUS is used

864C:ldy &36

Exit if the string is of zero length - as might happen in a macro call, for example

864E:beq &8672

Decrement Y, to make it be the offset from the start of the generated code to the last byte

8650:dey

Get the current byte from the code buffer

8651:lda &29,Y

Check &39 to see if the data is a string

8654:bit &39

Skip the next instruction if the data is not in the string buffer

8656:bpl &865B

Get the data byte from the string buffer

8658:lda &600,Y

Save it at the code origin

865B:sta (&3A),Y

Increment P%

865D:inc &440

8660:bne &8665

8662:inc &441

Increment O% if it is in use

8665:bcc &866F

8667:inc &43C

866A:bne &866F

866C:inc &43D

Copy the byte counter back into A

866F:tya

Continue the process of transferring bytes until the counter reaches zero

8670:bne &8650

Exit

8672:rts

Routines for each instruction group

This series of routines divides the instructions into different groups according to the addressing modes allowed. Each group is then dealt with separately

Move on to &86B7 if the instruction code is greater than or equal to &22. This leaves us with those instructions whose codes lie between &1A and &21 inclusive - the branch instructions

8673:cpx #&22

8675:bcx &86B7

Evaluate the integer expression following the instruction

8677:jsr &8821

Set Y(LSB) and A(MSB) to (destination of jump)-(current code position)-1. This makes Y and A contain one more than the branch distance.

867A:clc

867B:lda &2A

867D:sbc &440

8680:tay

8681:lda &2B

8683:sbc &441

Set the carry flag if $Y > 1$. This instruction and the next two are used to subtract a further 1 from the branching distance, to allow for the length of the instruction, which is two bytes

8686:cpx #&1

Decrement Y

8688:dey

Subtract 1 from A if Y was zero before the DEY instruction

8689:sbc #&0

This section checks the MSB of the branching distance to see if it is out of range. If the MSB of the branching distance is zero, jump to &86B2, since this means that the branch is probably not out of range

868B:beq &86B2

Jump to &86AD if the MSB of the branching distance is &FF

868D:cmp #&FF

868F:beq &86AD

This is a bug. The idea is to check OPT to see if errors should be trapped, and ignore the error if they should not. The way this has been done is to shift OPT one position to the right and then see if it has become zero. Thus, this system will only ignore 'Out of range' errors if O% is not being used.

8691:lda &28

8693:lsr A

8694:beq &86A5

8696:brk
8697:01-
8698:4F-O
8699:75-u
869A:74-t
869B:20-
869C:6F-o
869D:66-f
869E:20-
869F:72-r
86A0:61-a
86A1:6E-n
86A2:67-g
86A3:65-e
86A4:00-

Move zero into Y

86A5:tay

Save Y as the displacement

86A6:sty &2A

Set the code length to 2 bytes

86A8:ldy #&2

Join the original code

86AA:jmp &862B

Place the LSB of the displacement in A

86AD:tya

Use as the instruction's displacement if it is not out of range

86AE:bmi &86A6

86B0:bpl &8691

Place the LSB of the displacement in A

86B2:tya

Use as the instruction's displacement if it is not out of range

86B3:bpl &86A6

86B5:bmi &8691

Move on if the instruction code is greater than or equal to &29. This leaves us with the AND, EOR, ORA, ADC, CMP, LDA and SBC instructions. These are the arithmetic instructions, with the exception of STA, because STA does not allow the immediate addressing mode.

86B7:cpx #&29

86B9:bcs &86D3

Get the next character

86BB:jsr &8A97

See if it was a '#'

86BE:cmp #&23

Skip to the next addressing mode if not

86C0:bne &86DA

Add eight to the base op-code, to make it correct for immediate addressing

86C2:jsr &882F

Evaluate the expression following the '#'

86C5:jsr &8821

Set the flags according to the second byte of the expression

86C8:lda &2B

If the second byte was zero, branch back to indicate the correct length and exit, otherwise give a 'Byte' error message. This is to ensure we don't write 'LDA #257'

86CA:beq &86A8

86CC:brk

86CD:02-

86CE:42-B

86CF:79-y

86D0:74-t

86D1:65-e

86D2:00-

Move on if the instruction code is not that of STA

86D3:cpx #&36

86D5:bne &873F

Get the next character

86D7:jsr &8A97

Skip forwards if it is not a '('

86DA:cmp #&28

86DC:bne &8715

Evaluate the integer expression following the bracket

86DE:jsr &8821

Get the next character

86E1:jsr &8A97

Skip forwards if it is not a ')'

86E4:cmp #&29

86E6:bne &86FB

Get the next character

86E8:jsr &8A97

Give an error if it is not a comma

86EB:cmp #&2C

86ED:bne &870D

Add 16 to the base op-code

86EF:jsr &882C

Get the next character

86F2:jsr &8A97

Give an 'Index' error message if the character is not 'Y'

86F5:cmp #&59

86F7:bne &870D

Go back to assemble the instruction

86F9:beq &86C8

Give an 'Index' error if the next character is not a comma

86FB:cmp #&2C

86FD:bne &870D

Get the next character

86FF:jsr &8A97

Give an 'Index' error if it is not 'X'

8702:cmp #&58

8704:bne &870D

Get the next character

8706:jsr &8A97

Go back to assemble the instruction if it is a ')', otherwise fall into the 'Index' error message

8709:cmp #&29

870B:beq &86C8

870D:brk

870E:03-

870F:49-I

8710:6E-n

8711:64-d

8712:65-e

8713:78-x

8714:00-

Decrement the pointer to allow for the opening bracket being absent

8715:dec &A

Evaluate the integer expression following the instruction

8717:jsr &8821

Get the next character

871A:jsr &8A97

Skip if it was not a comma

871D:cmp #&2C

871F:bne &8735

Add 16 to the op-code

8721:jsr &882C

Get the next character

8724:jsr &8A97

Skip forwards if it is 'X'

8727:cmp #&58

8729:beq &8735

Give an 'Index' error if is not 'Y'

872B:cmp #&59

872D:bne &870D

Add eight to the op-code

872F:jsr &882F

Go on to indicate instruction length of three and exit

8732:jmp &879A

Add four to the op-code

8735:jsr &8832

Check for a 16 bit operand

8738:lda &2B

873A:bne &872F

Otherwise, assume the operand is eight bit

873C:jmp &86A8

Ignore all instructions whose code is greater than or equal to &2F. This leaves us with ASL, LSR, ROL, ROR, DEC and INC

873F:cpx #&2F

8741:bcs &876E

Skip the accumulator addressing mode if the instruction is INC or DEC

8743:cpx #&2D

8745:bcs &8750

Get the next character

8747:jsr &8A97

Goto &8767 if it is an 'A'

874A:cmp #&41

874C:beq &8767

Decrement the pointer offset to account for not finding the 'A'

874E:dec &A

Evaluate the integer expression following the instruction

8750:jsr &8821

Get the next character

8753:jsr &8A97

Branch to &8738 if it is not a comma

8756:cmp #&2C

8758:bne &8738

Add 16 to the op-code

875A:jsr &882C

Get the next character

875D:jsr &8A97

Go back if it was an 'X'

8760:cmp #&58

8762:beq &8738

Otherwise, give an 'Index' error message

8764:jmp &870D

Add four to the op-code

8767:jsr &8832

Set the instruction length to one byte and exit

876A:ldy #&1

876C:bne &879C

Ignore everything but CPX, CPY and BIT

876E:cpx #&32

8770:bcs &8788

Skip onwards if instruction is BIT, since it does not allow immediate addressing

8772:cpx #&31

8774:beq &8782

Get the next character

8776:jsr &8A97

Branch to &8780 if it is not a '#'

8779:cmp #&23

877B:bne &8780

Go back to join to the immediate addressing mode code

877D:jmp &86C5

Decrement the offset to account for not finding the hash character

8780:dec &A

Get the integer expression following the instruction

8782:jsr &8821

Join up with the previous code

8785:jmp &8735

Goto &8797 if the instruction is JSR

8788:cpx #&33

878A:beq &8797

Go on to &87B2 if the instruction is not JMP

878C:bcs &87B2

Get the next character

878E:jsr &8A97

Goto &879F if it is an '('

8791:cmp #&28

8793:beq &879F

Otherwise, decrement the pointer to account for not finding the bracket

8795:dec &A

Evaluate the integer expression following the instruction

8797:jsr &8821

Set the instruction length to three bytes and exit

879A:ldy #&3

879C:jmp &862B

Add 32 to the op-code

879F:jsr &882C

87A2:jsr &882C

Evaluate the integer expression following the instruction

87A5:jsr &8821

Get the next character

87A8:jsr &8A97

Ensure the ')' is present

87AB:cmp #&29

87AD:beq &879A

Give an 'Index' error if it is not

87AF:jmp &870D

Ignore all instructions except LDX, LDY, STA, STX and STY

87B2:cpx #&39

87B4:bcx &8813

Get the LSB of the mnemonic

87B6:lda &3D

Invert the bottom bit

87B8:eor #&1

Ignore all but the bottom five bits, which allows us to check on the destination of the instruction as A, X or Y

87BA:and #&1F

Save the last letter

87BC:pha

Jump forwards if the instruction is STX or STY, since they do not allow immediate addressing

87BD:cpx #&37

87BF:bcx &87F0

Get the next character

87C1:jsr &8A97

Skip forwards if it is not a '#'

87C4:cmp #&23

87C6:bne &87CC

Get the destination register back; we don't need to use it, but we don't want to clutter the stack up

87C8:pla

Rejoin the previous code

87C9:jmp &86C5

Decrement the offset to account for not finding the '#'

87CC:dec &A

Evaluate the integer expression following the instruction

87CE:jsr &8821

Get the destination register back off the stack

87D1:pla

Save it

87D2:sta &37

Get the next character

87D4:jsr &8A97

Skip forwards if it is a comma

87D7:cmp #&2C

87D9:beq &87DE

Go back to decide whether to use zero page addressing or not

87DB:jmp &8735

Get the next character

87DE:jsr &8A97

Ignore all but the bottom five bits

87E1:and #&1F

Check the indexing register against the destination register
87E3:cmp &37
 Skip forwards if they are different
87E5:bne &87ED
 Add 16 to the op-code
87E7:jsr &882C
 Join the original code
87EA:jmp &8735
 Give an 'Index' error message
87ED:jmp &870D
 Evaluate the integer expression following STX or STY
87F0:jsr &8821
 Get the source register back from the stack
87F3:pla
 Save it
87F4:sta &37
 Get the next character
87F6:jsr &8A97
 Exit if it is not a comma
87F9:cmp #&2C
87FB:bne &8810
 Get the next character
87FD:jsr &8A97
 Ignore all but the bottom five bits
8800:and #&1F
 Check it against the source register
8802:cmp &37
 Give an 'Index' error if they are different
8804:bne &87ED
 Add 16 to the op-code
8806:jsr &882C
 Make sure that the argument is eight bit
8809:lda &2B
880B:beq &8810
 Give a 'Byte' error message if it is not
880D:jmp &86CC
 Jump to rejoin the previous code
8810:jmp &8738
 Move to &883A for EQU
8813:bne &883A
 Evaluate the integer expression following OPT
8815:jsr &8821
 Set OPT according to this value
8818:lda &2A
881A:sta &28

Set the code length to zero

881C:ldy #&0

Rejoin the original code

881E:jmp &862B

Evaluate integer expression

This routine is used by the assembler to evaluate an integer expression, copying PTR#2 back to PTR#1

Evaluate the expression

8821:jsr &9B1D

Ensure it is integer

8824:jsr &92F0

Copy the offset of PTR#2 to PTR#1

8827:ldy &1B

8829:sty &A

Exit

882B:rts

Add 16 to the op-code

Fall the routine to add eight to the op-code

882C:jsr &882F

Fall into the routine to add another eight

Add eight to the op-code

Call the routine to add four to the op-code

882F:jsr &8832

Fall into the routine to add another four

Add four to the op-code

Get the current op-code

8832:lda &29

Add four to it

8834:clc

8835:adc #&4

Save it as the new op-code

8837:sta &29

Exit

8839:rts

This routine deals with the EQU pseudo-instruction

Set the length of the code generated by EQU to one byte

883A:ldx #&1

Get the character after the letters EQU

883C:ldy &A

883E:inc &A**8840:lda (&B),Y**

Jump forwards if it is the letter 'B'

8842:cmp #&42**8844:beq &8858**

Make the length be two bytes (for EQUW)

8846:inx

Jump forwards if the letter is 'W'

8847:cmp #&57**8849:beq &8858**

Make the length be four bytes (for EQUW)

884B:ldx #&4

Jump forwards if the character is 'D'

884D:cmp #&44**884F:beq &8858**

Skip forwards if it is 'S', for EQUW

8851:cmp #&53**8853:beq &886A**

Jump to 'Syntax error' if anything else follows 'EQU'

8855:jump &982A

Save the number of bytes to be generated on the stack

8858:txa**8859:pha**

Evaluate the expression

885A:jsr &8821

Copy the IAC to &29,A,B,C

885D:ldx #&29**885F:jsr &BE44**

Get the length of code back into Y

8862:pla**8863:tay**

Go back to insert the code

8864:jump &862B

Direct jump to the 'Type mismatch' error message

8867:jump &8C0E

Save the current OPT value (this is in case EQUW is used to implement macros)

886A:lda &28**886C:pha**

Evaluate the expression

886D:jsr &9B1D

Give a 'Type mismatch' if it is not string

8870:bne &8867

Restore OPT

8872:pla**8873:sta &28**

Update PTR #1

8875:jsr &8827

Set the code length to &FF to indicate that the code length is in &36

8878:ldy #&FF

Exit

887A:bne &8864

Close up Y bytes

This routine is used by the tokenisation process. It operates on a string of characters terminated by a carriage return. The pointer at (&37) points to some point in this string. This routine substitutes a single byte (A) for the Y bytes pointed to by the pointer. So, for example, if the pointer pointed to the word 'ENVELOPE', this routine would be called with A=&E2 and Y=8

Save the byte to be inserted

887C:pha

Clear the carry flag for the forthcoming addition

887D:clc

Get the number of bytes to be removed into A

887E:tya

Add this to the LSB of the pointer. This will give us the location from where bytes should be moved down

887F:adc &37

Save the answer as the LSB of a new pointer in (&39)

8881:sta &39

Zero Y for the forthcoming block move

8883:ldy #&0

Also zero A, for the MSB calculation of the second pointer

8885:tya

Add zero and the carry flag to the MSB of the original pointer

8886:adc &38

Save this as the MSB of the new pointer

8888:sta &3A

Get the byte to be inserted back from the stack

888A:pla

Save this at the destination pointer

888B:sta (&37),Y

Increment Y to make sure the inserted byte is not overwritten

888D:iny

Get a byte from the source pointer

888E:lda (&39),Y

Place it at the destination pointer

8890:sta (&37),Y

Continue this process until a carriage return is encountered

8892:cmp #&D

8894:bne &888D

Then exit

8896:rts

Convert ASCII number to 16 bit binary

This routine converts the ASCII number pointed to by (&37) to a binary number in &3D and &3E. Y is set to the length of the number. The routine is used to encode line numbers. On entry, A contains the first digit and Y contains zero

This routine falls straight into the routine which inserts the number in binary into the text, and so cannot be used directly in your own programs

Convert the first digit to a binary number by ignoring all but the bottom four bits (the top nybble will always contain 3)

8897:and #&F

Save this digit as the LSB of the number

8899:sta &3D

Use Y to zero the MSB

889B:sty &3E

Get the next character

889D:iny

889E:lda (&37),Y

Skip ahead if the character is too big to be a digit

88A0:cmp #&3A

88A2:bcs &88DA

Or if it is too small to be a digit

88A4:cmp #&30

88A6:bcc &88DA

Otherwise, convert it to a binary digit

88A8:and #&F

Save it on the stack

88AA:pha

Now we have to multiply &3D,&3E by 10

Save the original MSB in X

88AB:ldx &3E

Get the LSB into A

88AD:lda &3D

Multiply the LSB by two

88AF:asl A

Multiply the MSB by two

88B0:rol &3E

Exit if the number is now over 65535

88B2:bmi &88D5

Multiply the number by two again - this will mean that it has been multiplied by four altogether

88B4:asl A

88B5:rol &3E

Check for overflow

88B7:bmi &88D5

Add the current LSB to the original LSB. This makes a total multiplication of five

88B9:adc &3D

Save this new LSB in &3D

88BB:sta &3D

Add the original MSB to the current MSB, leaving the answer in A

88BD:txa

88BE:adc &3E

Multiply the number by two again, giving a total multiplication of 10

88C0:asl &3D

88C2:rol A

Check for overflow

88C3:bmi &88D5

88C5:bcs &88D5

Save the MSB

88C7:sta &3E

Get back the new digit

88C9:pla

Add it to the number

88CA:adc &3D

88CC:sta &3D

Return to get a new digit if the MSB does not need to be adjusted

88CE:bcc &889D

Adjust the MSB and go back to get a new digit unless the number is now over 32767

88D0:inc &3E

88D2:bpl &889D

These two instructions are dummies - they are used to discard any digit left on the stack if a premature exit has to be made from the main loop

88D4:pha

88D5:pla

Indicate overflow by setting the carry flag, and leave the length of the number as zero

88D6:ldy #&0

88D8:sec

Exit

88D9:rts

Make Y reflect the length of the line number

88DA:dey

Insert line number into buffer

This routine is entered with Y containing the length of the ASCII line number at (&37) with the binary equivalent of the number in (&3D)

Get the token for a line number

88DB:lda #&8D

Insert it into the text of the buffer, substituting it for the entire line number

88DD:jsr &887C

Make a new pointer at (&39) that points three locations higher than the pointer at (&37). Bear in mind that the routine at &887C always exits with the carry flag set

88E0:lda &37**88E2:adc #&2****88E4:sta &39****88E6:lda &38****88E8:adc #&0****88EA:sta &3A**

Y is currently pointing to the carriage return at the end of the line

Get a byte from the low pointer

88EC:lda (&37),Y

Save it at the high pointer

88EE:sta (&39),Y

Continue this process for the remainder of the line, thus making a three byte space after the line number token

88F0:dey**88F1:bne &88EC**

Set Y to three, to indicate that the third byte of the line number is being inserted into the text

88F3:ldy #&3

Get the MSB of the line number

88F5:lda &3E

Make sure bit 6 is set - this will ensure that the third byte is in the ASCII range

88F7:ora #&40

Save this as the third byte of the binary representation

88F9:sta (&37),Y

Point to the second byte

88FB:dey

Get the LSB of the line number

88FC:lda &3D

Ignore all but the bottom six bytes

88FE:and #&3F

Ensure that bit 6 is set, to make sure the byte is in the ASCII range

8900:ora #&40

Save this as the second byte

8902:sta (&37),Y

Point to the first byte of the number

8904:dey

Mask out all but the top two bits of the LSB of the line number

8905:lda &3D**8907:and #&C0****8909:sta &3D**

Mask the MSB in the same way

890B:lda &3E

890D:and #&C0

Divide the MSB by four, to make the two bits we are interested in become bits 4 and 5

890F:lsr A

8910:lsr A

Include the two bits from the LSB

8911:ora &3D

Divide by four again, to free the top two bits of the byte

8913:lsr A

8914:lsr A

Make sure that bit six is set

8915:eor #&54

Save this as the first byte

8917:sta (&37),Y

Add three to the pointer, to make the pointer point to the last byte of the line number

8919:jsr &8944

891C:jsr &8944

891F:jsr &8944

Make the offset be zero

8922:ldy #&0

Indicate success by clearing the carry flag

8924:clc

Exit

8925:rts

Test A for an alphanumeric character

This routine tests A to see if it contains an alphanumeric character. The carry flag is set on exit if the character is alphanumeric, and cleared otherwise. It is easier to understand this routine if you refer to the ASCII table in the User Guide

Fail if A> = &7B ('{')

8926:cmp #&7B

8928:bcs &8924

Succeed if A> = &5F ('_')

892A:cmp #&5F

892C:bcs &893C

Fail if A> = &5B ('[')

892E:cmp #&5B

8930:bcs &8924

Succeed if A> = &41 ('A')

8932:cmp #&41

8934:bcs &893C

Fail if A> = &3A (':')

8936:cmp #&3A

8938:bcs &8924

Check to see if the character is less than '0'

893A:cmp #&30

893C:rts

Success if the character is a period, possible failure otherwise

893D:cmp #&2E

893F:bne &8936

8941:rts

Get a character and increment pointer

This routine gets a character from the current setting of (&37), and then increments the pointer. Most of the time, it is entered at &8944 - only the error handling routine uses the first instruction

Get a character

8942:lda (&37),Y

Increment the LSB of the pointer

8944:inc &37

If it has not become zero, exit

8946:bne &894A

Increment the MSB of the pointer

8948:inc &38

Exit

894A:rts

Increment pointer and get a character

This routine is like the last one, except that the pointer is incremented before the character is accessed

894B:jsr &8944

894E:lda (&37),Y

8950:rts

Tokenise a line of text

This routine tokenises a line of text pointed to by &37,&38 and terminated by a carriage return

Locations &3B and &3C are used as flags. The routine sets their default values in the first three instructions, but it is usually called at &8957, when the flags are set in advance

The flag at &3B is used to determine whether the pointer is at the start of a statement or not. This is necessary because of the different action of some keywords depending on their positions. For example, an asterisk at the start of a line should make the computer stop tokenising - otherwise commands like *SAVE would not work. On the other hand, an asterisk as in 'F=2*B' should not make the line be abandoned. &3B contains 0 at the start of a statement, and a non-zero number at other times.

The flag at &3C is used to determine whether or not numbers should be tokenised into line numbers. If it is zero, numbers will not be tokenised, otherwise they will.

Set both flags to zero, the default condition. These settings are used for the EVAL function, which has to tokenise its argument

8951:ldy #&0

8953:sty &3B

8955:sty &3C

Get a character

8957:lda (&37),Y

Exit if it is a carriage return

8959:cmp #&D

895B:beq &894A

Skip forwards if it is not a space

895D:cmp #&20

895F:bne &8966

Increment the pointer past the space

8961:jsr &8944

Go back to get another character

8964:bne &8957

Skip the next section if the current character is not an ampersand. The next bit of code increments the pointer past a hexadecimal number

8966:cmp #&26

8968:bne &897C

Increment the pointer and get a character

896A:jsr &894B

Check if it is a digit

896D:jsr &8936

Go back and check the next character if it is a digit.

8970:bcs &896A

Go back to the beginning if the character is below 'A', since the number has come to an end

8972:cmp #&41

8974:bcc &8957

Continue scanning the number if it is less than or equal to 'F'

8976:cmp #&47

8978:bcc &896A

Otherwise, return to the start to get another character

897A:bcs &8957

Skip this section if the character is not a double quote

897C:cmp #&22

897E:bne &898C

Increment the pointer and get a character

8980:jsr &894B

Get out of quote mode if it is another double quote

8983:cmp #&22

8985:beq &8961

Continue until a carriage return is found

8987:cmp #&D**8989:bne &8980**

Exit

898B:rts

Skip to the next section if the character is not a colon

898C:cmp #&3A**898E:bne &8996**

Zero &3B to show that the pointer is at the start of a statement

8990:sty &3B

Zero &3C to allow line numbers to be tokenised

8992:sty &3C

Go back to get another character

8994:beq &8961

Return to the start if the character is a comma

8996:cmp #&2C**8998:beq &8961**

Skip this section if the character is not an asterisk

899A:cmp #&2A**899C:bne &89A3**

If the pointer is not at the start of the line, set the flags and go back to the start of the routine

899E:lda &3B**89A0:bne &89E3**

Otherwise, abort the routine, since the rest of the line is an operating system command

89A2:rts

Branch forwards if the character is a period

89A3:cmp #&2E**89A5:beq &89B5**

Check the character for a number

89A7:jsr &8936

Skip forwards if it is not a number

89AA:bcc &89DF

If numbers are not to be tokenised, skip the next section

89AC:ldx &3C**89AE:beq &89B5**

Tokenise the number

89B0:jsr &8897

Return to the start if the tokenisation was successful

89B3:bcc &89E9

Get the current character

89B5:lda (&37),Y

Check for a number or a period

89B7:jsr &893D

Set both flags and return to the start if the number has finished

89BA:bcc &89C2

Increment the pointer

89BC:jsr &8944

Continue scanning to the end of the number

89BF:jmp &89B5

Set both flags

89C2:ldx #&FF

89C4:stx &3B

89C6:sty &3C

Return to the start

89C8:jmp &8957

Check if the character is a digit

89CB:jsr &8926

Return to the start if it is not

89CE:bcc &89E3

Get the current character

89D0:ldy #&0

89D2:lda (&37),Y

Check whether it is an alphanumeric character

89D4:jsr &8926

Return to the start and set both flags if it is not

89D7:bcc &89C2

Increment the pointer past the current character of the variable name

89D9:jsr &8944

Continue until the end of the name is reached

89DC:jmp &89D2

Jump forwards if the character is greater than or equal to 'A'

89DF:cmp #&41

89E1:bcs &89EC

Set both the flags

89E3:ldx #&FF

89E5:stx &3B

89E7:sty &3C

Return to the start

89E9:jmp &8961

Just scan to the end of the name if the character is greater than 'X'

89EC:cmp #&58

89EE:bcs &89CB

Make a pointer at (&39) point to &8071, the start of the keyword table

89F0:ldx #&71

89F2:stx &39

89F4:ldx #&80

89F6:stx &3A

Compare the current character with the first character of the current keyword

89F8:cmp (&39),Y

Skip past the rest of the variable name if the current character is less than the first character of the current keyword - remember that the keywords are in alphabetical order

89FA:bcc &89D2

Move to the next keyword if there was no match

89FC:bne &8A0D

Point to the next character of the keyword

89FE:iny

Get the keyword character from the ROM

89FF:lda (&39),Y

If it is a token, the end of the keyword has been reached, and we have a match, so go to &8A37 to do the tokenising

8A01:bmi &8A37

Compare this character with one from RAM

8A03:cmp (&37),Y

Continue the search if there was a match

8A05:beq &89FE

Get the RAM character that didn't match

8A07:lda (&37),Y

Goto &8A18 if it was a period, since this means that we are dealing with an abbreviation

8A09:cmp #&2E

8A0B:beq &8A18

Get the next ROM character

8A0D:iny

8A0E:lda (&39),Y

Continue if we haven't reached the token marking the end of the keyword

8A10:bpl &8A0D

Check the token found against &FE, which is the last token in the list

8A12:cmp #&FE

If we have not reached the end of the list, goto &8A25

8A14:bne &8A25

Otherwise, go back to skip the rest of the variable name

8A16:bcs &89D0

Get the next ROM character

8A18:iny

8A19:lda (&39),Y

If it is a token, skip

8A1B:bmi &8A37

Increment the ROM pointer and continue looking for the token. Notice that this method does not disturb Y, which must continue to have the length of the keyword as it was found in RAM, which will be a different length to normal if an abbreviation is used


```

8A1D:inc &39
8A1F:bne &8A19
8A21:inc &3A
8A23:bne &8A19

```

Make the pointer point to the next keyword by adding Y + 2 to it

```

8A25:sec
8A26:iny
8A27:tya
8A28:adc &39
8A2A:sta &39
8A2C:bcc &8A30
8A2E:inc &3A

```

Get the first character of the RAM keyword

```

8A30:ldy #&0
8A32:lda (&37),Y

```

Go back to look at the current ROM keyword

```

8A34:jmp &89F8

```

If we get here, A contains the correct token

Save the token in X

```

8A37:tax

```

Increment the ROM pointer, and get the tokenising byte

```

8A38:iny
8A39:lda (&39),Y

```

Save the tokenising byte in &3D

```

8A3B:sta &3D

```

Move the pointer back to make it point to the character after the RAM keyword

```

8A3D:dey

```

Move bit 0 of the tokenising byte into the carry flag

```

8A3E:lsr A

```

Skip if the bit is clear

```

8A3F:bcc &8A48

```

Get the next RAM character

```

8A41:lda (&37),Y

```

Check whether it is alphanumeric

```

8A43:jsr &8926

```

Stop tokenising if it is alphanumeric

```

8A46:bcs &89D0

```

Copy the token into A

```

8A48:txa

```

Skip to &8A54 if the tokenising byte does not have bit 6 set

```

8A49:bit &3D

```

```

8A4B:bvc &8A54

```

Skip to &8A54 if the pointer is not at the start of a statement

8A4D:ldx &3B

8A4F:bne &8A54

Add &40 to the token

8A51:clc

8A52:adc #&40

Point to the last character of the keyword

8A54:dey

Insert the token in the text

8A55:jsr &887C

Set X and Y to make it easier to alter the settings of the flags

8A58:ldy #&0

8A5A:ldx #&FF

Get the tokenising byte

8A5C:lda &3D

Skip if bit 1 is not set

8A5E:lsr A

8A5F:lsr A

8A60:bcc &8A66

Indicate that the pointer is not at the start of a line

8A62:stx &3B

Indicate that numbers should not be tokenised

8A64:sty &3C

Skip this section if bit 2 of the tokenising byte is not set

8A66:lsr A

8A67:bcc &8A6D

Indicate that the pointer is at the start of a statement and that numbers should not be tokenised

8A69:sty &3B

8A6B:sty &3C

Skip this section if bit 3 of the tokenising byte is not set

8A6D:lsr A

8A6E:bcc &8A81

Save the tokenising byte

8A70:pha

Make Y contain 1

8A71:iny

Get the character

8A72:lda (&37),Y

Check if it is alphanumeric

8A74:jsr &8926

Go forwards if it is not

8A77:bcc &8A7F

Increment the pointer past the character

8A79:jsr &8944

Go back till the end of the alphanumeric string is reached

8A7C:jmp &8A72

Decrement Y to make it zero again

8A7F:dey

Restore the tokenising byte

8A80:pla

Skip this section if bit 4 of the tokenising byte is not set

8A81:lsr A

8A82:bcc &8A86

Indicate that numbers should be tokenised

8A84:stx &3C

Return immediately if bit 5 of the tokenising byte is set

8A86:lsr A

8A87:bcs &8A96

Otherwise, go back to the start

8A89:jmp &8961

Get next character from PTR #2

This routine gets the next character from PTR #2, ignoring spaces

Get the offset of PTR #2

8A8C:ldy &1B

Point to the next character

8A8E:inc &1B

Get the current character

8A90:lda (&19),Y

Go back if it was a space

8A92:cmp #&20

8A94:beq &8A8C

Exit

8A96:rts

Get next character from PTR #1

This routine gets the next character from PTR #1, ignoring spaces

Get the offset of PTR #1

8A97:ldy &A

Point to the next character

8A99:inc &A

Get the current character

8A9B:lda (&B),Y

Go back if it was a space

8A9D:cmp #&20

8A9F:beq &8A97

Exit

```

8AA1:rts
8AA2:brk
8AA3:05-
8AA4:4D-M
8AA5:69-i
8AA6:73-s
8AA7:73-s
8AA8:69-i
8AA9:6E-n
8AAA:67-g
8AAB:20-
8AAC:2C-,
8AAD:00-
    
```

Check for comma at PTR #2

Get next character

```
8AAE:jsr &8A8C
```

Give 'Missing,' error if it is not a comma

```
8AB1:cmp #&2C
```

```
8AB3:bne &8AA2
```

Exit

```
8AB5:rts
```

OLD command routine

This routine replaces the &FF end of program marker at PAGE + 1 with zero, and then checks for a 'Bad program'

Check for the end of the statement

```
8AB6:jsr &9857
```

Make (&37) point to PAGE

```
8AB9:lda &18
```

```
8ABB:sta &38
```

```
8ABD:lda #&0
```

```
8ABF:sta &37
```

At this point, Y always contains 1, so this line places a zero byte at PAGE + 1

```
8AC1:sta (&37),Y
```

Check for 'Bad program'

```
8AC3:jsr &BE6F
```

Do a warm start

```
8AC6:bne &8AF3
```

END statement routine

Check for the end of the statement

```
8AC8:jsr &9857
```

Check for 'Bad program'

8ACB:jsr &BE6F

Do a warm start

8ACE:bne &8AF6

STOP statement routine

Check for the end of the statement

8AD0:jsr &9857

Fall into the 'STOP' error message. For some reason, the STOP token has not been used in the error message

8AD3:brk

8AD4:00-

8AD5:53-S

8AD6:54-T

8AD7:4F-O

8AD8:50-P

8AD9:00-

NEW command routine

Check for the end of the statement

8ADA:jsr &9857

Fall into a cold start

Cold start

Entering BASIC here does an almost complete re-initialization. The system does everything except set the default value of PAGE and seed the RND store

Prepare the accumulator with a carriage return

8ADD:lda #&D

Make TOP be the same as PAGE

8ADF:ldy &18

8AE1:sty &13

8AE3:ldy #&0

8AE5:sty &12

Turn TRACE off

8AE7:sty &20

Place a carriage return at PAGE

8AE9:sta (&12),Y

Place &FF at PAGE + 1

8AEB:lda #&FF

8AED:iny

8AEE:sta (&12),Y

Update TOP to PAGE + 2

8AF0:iny

8AF1:sty &12

Warm start

Starting BASIC from here will ensure that the current program is not cleared. Sometimes, a warm start is taken from &8AF6, in which case the stacks are not cleared

Clear variables and various stacks

8AF3:jsr &BD20

Set PTR #1 to &700, the keyboard buffer

8AF6:ldy #&7

8AF8:sty &C

8AFA:ldy #&0

8AFC:sty &B

Set the error vector to &B433

8AFE:lda #&33

8B00:sta &16

8B02:lda #&B4

8B04:sta &17

Get a line of text to &700, using '>' as a prompt

8B06:lda #&3E

8B08:jsr &BC02

Reset the error vector

8B0B:lda #&33

8B0D:sta &16

8B0F:lda #&B4

8B11:sta &17

Set OPT to &FF, to make sure 'No such variable' error messages are issued correctly

8B13:ldx #&FF

8B15:stx &28

Indicate to the tokenising routine that line numbers should be tokenised. This is so that the line number at the start of a program line is tokenised

8B17:stx &3C

Set the machine stack to &FF

8B19:txs

Initialise stacks

8B1A:jsr &BD3A

A always contains zero here; after this instruction, Y will as well

8B1D:tay

Make the tokenising pointer point to the keyboard buffer

8B1E:lda &B

8B20:sta &37

8B22:lda &C

8B24:sta &38

Indicate to the tokenising routine that the pointer is at the start of a statement, to ensure that asterisk commands work correctly

8B26:sty &3B

Set the PTR #1 offset to zero

8B28:sty &A

Tokenise the keyboard buffer

8B2A:jsr &8957

See if the line started with a line number

8B2D:jsr &97DF

Skip if there was no line number; this must be an immediate command

8B30:bcc &8B38

Insert the line into the program

8B32:jsr &BC8D

Do a warm start

8B35:jmp &8AF3

Get a character from the line

8B38:jsr &8A97

Goto &8BB1 if the character was greater than or equal to &C6 (which indicates that it was a token for a command or statement)

8B3B:cmp #&C6

8B3D:bcs &8BB1

Otherwise, go to &8BBF to interpret an assignment statement

8B3F:bcc &8BBF

Direct jump to the warm start point

8B41:jmp &8AF6

Direct jump to the assembler entry point

8B44:jmp &8504

= < expression > routine

Control passes here when an equals sign is found as the first character of a statement

Get the current stack pointer

8B47:tsx

Give a 'No FN' error message if it is greater than &FC. If the stack pointer is greater than &FC, less than four bytes are on the stack. If a function is being executed, there must be at least four bytes on the stack, the return address of the function, the number of parameters/local variables and the function indicator byte

8B48:cpx #&FC

8B4A:bcs &8B59

Get the byte at the top (or bottom, depending on your terminology) of the stack

8B4C:lda &1FF

Make sure that it tells us we are in a function

8B4F:cmp #&A4

8B51:bne &8B59

Evaluate the expression following the equals sign

8B53:jsr &9B1D

Check for the end of the line, then execute a RTS instruction to take us back to the place where the function was called

8B56:jmp &984C

This error message reads 'No FN' - the letters 'FN' have been replaced with their token

8B59:brk
 8B5A:07-
 8B5B:4E-N
 8B5C:6F-o
 8B5D:20-
 8B5E:A4-\$
 8B5F:00-

Get the offset of PTR # 1

8B60:ldy &A

Decrement the pointer, to point to the character which baffled the assignment routine

8B62:dey

Get the character

8B63:lda (&B),Y

Check for '='

8B65:cmp #&3D

8B67:beq &8B47

Check for '*'

8B69:cmp #&2A

8B6B:beq &8B73

Check for '['

8B6D:cmp #&5B

8B6F:beq &8B44

If it was none of these, give a rather devious 'Mistake' error message

8B71:bne &8B96

'*' statement routine

This routine handles all operating system commands

Make PTR # 1 point directly to the character after the asterisk

8B73:jsr &986D

Make X,Y point to the same place

8B76:ldx &B

8B78:ldy &C

Call OSCLI to execute the command

8B7A:jsr &FFF7

Fall into routine to skip to the end of the line

DATA/DEF/REM statement routine

This routine deals with all these statements by moving PTR # 1 to the start of the next statement

Set A to a carriage return, since we now have to skip the rest of the line till we find a carriage return

8B7D:lda #&D

Get the offset from PTR # 1

8B7F:ldy &A

Decrement it to allow for the first INY instruction

8B81:dey

Point to the next character

8B82:iny

Compare it with a carriage return

8B83:cmp (&B),Y

Continue searching if there was no match

8B85:bne &8B82

Fall into the routine to execute the next statement

Execute next statement routine

Check for ELSE, and skip to the end of the line if it is present

8B87:cmp #&8B

8B89:beq &8B7D

Check for immediate mode

8B8B:lda &C

8B8D:cmp #&7

Do a warm start if so

8B8F:beq &8B41

Move PTR#1 to the next statement

8B91:jsr &9890

Branch always to execute the next statement

8B94:bne &8BA3

Move PTR#1 back one position

8B96:dec &A

Check for the end of the statement

8B98:jsr &9857

Get the character following the statement

8B9B:ldy #&0

8B9D:lda (&B),Y

Check for ELSE and carriage return if it is not a colon

8B9F:cmp #&3A

8BA1:bne &8B87

Skip the spaces preceding the statement

8BA3:ldy &A

8BA5:inc &A

8BA7:lda (&B),Y

8BA9:cmp #&20

8BAB:beq &8BA3

If the first character is not a token for a statement, enter the assignment routine

8BAD:cmp #&CF

8BAF:bcc &8BBF

Get the token into X

8BB1:tax

Get the LSB of the action address of the keyword

8BB2:lda &82DF,X

Save it

8BB5:sta &37

Get the MSB of the action address of the keyword

8BB7:lda &8351,X

Save it

8BBA:sta &38

Jump to execute the statement/command/function

8BBC:jmp (&37)

Statement not found

Control passes here if a statement does not start with a token for a command or a statement. This means that the statement is either an assignment statement, or it is one of the things tested for in the routine at &8B60

Copy PTR # 1 to PTR # 2

8BBF:ldx &B

8BC1:stx &19

8BC3:ldx &C

8BC5:stx &1A

8BC7:sty &1B

Look for a variable name

8BC9:jsr &95DD

Jump into LET if the name of an existing variable was found

8BCC:bne &8BE9

If a bad variable name was found, go back to check for the other things

8BCE:bcs &8B60

Save the offset of PTR # 2

8BD0:stx &1B

Check for an '=' at PTR # 2

8BD2:jsr &9841

Create the variable

8BD5:jsr &94FC

Set the number of bytes to be cleared as 5

8BD8:ldx #&5

Increment this to six if the type of the variable was real

8BDA:cpx &2C

8BDC:bne &8BDF

8BDE:inx

Clear the variable

8BDF:jsr &9531

Back up one position, because the offset of PTR # 1 was not adjusted after the spaces were skipped at &8BA5

8BE2:dec &A

LET statement routine

Get a variable name, creating a new variable if needed

8BE4:jsr &9582

Give a 'Syntax error' if a variable name was not found

8BE7:beq &8C0B

Move to &8BFB if the variable was not a string variable

8BE9:bcc &8BFB

Save the type and address of the variable

8BEB:jsr &BD94

Check for an '=' then evaluate the expression following it

8BEE:jsr &9813

Give a 'Type mismatch' error if the expression was not a string

8BF1:lda &27

8BF3:bne &8C0E

Assign the string

8BF5:jsr &8C1E

Exit

8BF8:jmp &8B9B

Save the address and type of the variable

8BFB:jsr &BD94

Check for an '=' and evaluate the expression following it

8BFE:jsr &9813

Give a 'Type mismatch' error if the expression was a string

8C01:lda &27

8C03:beq &8C0E

Assign the expression

8C05:jsr &B4B4

Exit

8C08:jmp &8B9B

Direct jump to 'Syntax error'

8C0B:jmp &982A

8C0E:brk

8C0F:06-

8C10:54-T

8C11:79-y

8C12:70-p

8C13:65-e

8C14:20-

8C15:6D-m

8C16:69-i

8C17:73-s

8C18:6D-m

8C19:61-a

8C1A:74-t

8C1B:63-c

8C1C:68-h**8C1D:00-****String assignments**

This routine assigns the string in the string buffer to the string variable whose address and type are stored on the BASIC stack

Get the address and type of the variable back off the stack

8C1E:jsr &BDEA

Get the type of the variable

8C21:lda &2C

Branch to &8CA2 if the variable is a string terminated in a carriage return

8C23:cmp #&80**8C25:beq &8CA2**

Get the allocated length of the string

8C27:ldy #&2**8C29:lda (&2A),Y**

Compare it with the length of the string in the string buffer

8C2B:cmp &36

Goto &8C84 if there is enough room

8C2D:bcs &8C84

Save the current setting of VARTOP

8C2F:lda &2**8C31:sta &2C****8C33:lda &3****8C35:sta &2D**

Get the length of the string in the buffer

8C37:lda &36

Jump forwards if it is less than eight

8C39:cmp #&8**8C3B:bcc &8C43**

Otherwise add eight to the length, to leave some room for growth

8C3D:adc #&7

If there was no overflow, jump forwards

8C3F:bcc &8C43

Give the string a length of 255 if there is not enough room

8C41:lda #&FF

Clear the carry flag ready for the next addition

8C43:clc

Save the new allocated length of the string

8C44:pha

Place this length in X

8C45:tax

Get the allocated length again

8C46:lda (&2A),Y

Add it to the LSB of the start address of the string

8C48:ldy #&0

8C4A:adc (&2A),Y

Exclusive-or this result with the LSB of VARTOP

8C4C:eor &2

Jump forwards if they are different

8C4E:bne &8C5F

Get the MSB of the start address and add it to zero

8C50:iny

8C51:adc (&2A),Y

Exclusive or it with the MSB of VARTOP

8C53:eor &3

Skip if they are different; if they are not different the variable we are investigating is at the top of the pile, and so can be extended freely

8C55:bne &8C5F

Zero &2D, which holds the MSB of VARTOP

8C57:sta &2D

Get the allocated length of the string

8C59:txa

Subtract previously allocated length

8C5A:iny

8C5B:sec

8C5C:sbc (&2A),Y

Get into the correct register

8C5E:tax

Get the length to be allocated on top of VARTOP into A

8C5F:txa

Add this length to the LSB of VARTOP

8C60:clc

8C61:adc &2

Save this result in Y

8C63:tay

Add the MSB

8C64:lda &3

8C66:adc #&0

Check if VARTOP is now greater than the stack pointer

8C68:cpy &4

8C6A:tax

8C6B:sbc &5

Give a 'No room' error if so

8C6D:bcs &8CB7

Save the new value of VARTOP

8C6F:sty &2

8C71:stx &3

Get back the allocated length of the new string

8C73:pla

Save this as the allocated length of the string

8C74:ldy #&2

8C76:sta (&2A),Y

Point to the MSB of the strings start address

8C78:dey

Get the flag which indicates whether the top variable is being extended or whether a whole new string space is being defined

8C79:lda &2D

Don't change the start address of the string if the string is just being extended from the same place

8C7B:beq &8C84

Save the MSB of the strings start address

8C7D:sta (&2A),Y

8C7F:dey

Save the LSB of the start address of the string

8C80:lda &2C

8C82:sta (&2A),Y

Point to the length byte of the string information block

8C84:ldy #&3

Get the length of the string

8C86:lda &36

Store it in the block

8C88:sta (&2A),Y

Return if the length is zero

8C8A:beq &8CA1

Point to the MSB of the start address

8C8C:dey

8C8D:dey

Get the MSB of the start address and store it

8C8E:lda (&2A),Y

8C90:sta &2D

Get the LSB of the start address and store it

8C92:dey

8C93:lda (&2A),Y

8C95:sta &2C

Get a character from the string buffer

8C97:lda &600,Y

Save it in the variable area

8C9A:sta (&2C),Y

Point to the next byte

8C9C:iny

Continue until the end of the string is met

8C9D:cpy &36

8C9F:bne &8C97

Exit

8CA1:rts

Assignments for defined address strings

Place 'return' at the end of the string to be assigned

8CA2:jsr &BEBA

If length is zero, skip to the end, inserting only a carriage return

8CA5:cpy #&0

8CA7:beq &8CB4

Get a character from the buffer

8CA9:lda &600,Y

Place it in memory

8CAC:sta (&2A),Y

Point to the next byte down

8CAE:dey

Continue this process until Y becomes zero

8CAF:bne &8CA9

Transfer the last byte

8CB1:lda &600

8CB4:sta (&2A),Y

Exit

8CB6:rts

8CB7:brk

8CB8:00-

8CB9:4E-N

8CBA:6F-o

8CBB:20-

8CBC:72-r

8CBD:6F-o

8CBE:6F-o

8CBF:6D-m

8CC0:00-

Unstack a parameter

When a procedure or a function terminates, the previous values of its LOCAL variables and parameters have to be restored. For example, if a procedure contains the line 'LOCAL A\$', a search is carried out for the variable A\$. If it does not exist, it is created. The value, address and type of the variable is then pushed on the BASIC stack. The procedure is then free to use the variable A\$. On exit, this routine restores the previous value of the local variables or parameters.

On entry, location &39 contains the type of the variable, (&37) contains the address of the variable and the value is on the BASIC stack. The type of the variable is stored using the same codes as the CALL statement.

Get the type of the item

8CC1:lda &39

Check if the item is a string at a defined address

8CC3:cmp #&80

Goto &8CEE if it is

8CC5:beq &8CEE

Goto &8D02 if the item is numeric - its type is less than 128

8CC7:bcc &8D03

Therefore, we are now dealing with a dynamic string. Unlike the LET statement, the string being assigned is bound to fit in the space allocated to it, so the process of assigning the string is considerably simpler. The first step is to get the length of the string from the top of the stack

8CC9:ldy #&0

8CCB:lda (&4),Y

Transfer the length to X

8CCD:tax

Skip transferring the characters in the string if it is null

8CCE:beq &8CE5

Get the LSB of the address of the string

8CD0:lda (&37),Y

Subtract one, to take account of the length of the string being stored on the stack as well as the text of the string

8CD2:sub #&1

Save this address in a new pointer at (&39)

8CD4:sta &39

Get the MSB of the string start address

8CD6:iny

8CD7:lda (&37),Y

Decrement it if necessary

8CD9:sub #&0

Save it. (&39) now points one byte below the destination address of the string

8CDB:sta &3A

Get a character from the stacked string

8CDD:lda (&4),Y

Save it in the variable area

8CDF:sta (&39),Y

Point to the next character

8CE1:iny

Decrement the remaining length of the string

8CE2:dex

Continue until the remaining length becomes zero

8CE3:bne &8CDD

Get the length of the string

8CE5:lda (&4,X)

Save this in the string information block

8CE7:ldy #&3

8CE9:sta (&37),Y

Jump into the routine which discards the string at the top of the stack and then exit

8CEB:jmp &BDDC

This section deals with strings at a fixed address terminated by a carriage return. The first step is to get the length of the string

8CEE:ldy #&0

8CF0:lda (&4),Y

Place the length in X

8CF2:tax

Skip moving the string if it has a length of zero

8CF3:beq &8CFF

Get the next character from the string on the stack

8CF5:iny

8CF6:lda (&4),Y

Decrement the pointer to allow for the length being stored on the stack

8CF8:dey

Save the current character

8CF9:sta (&37),Y

Make up for the DEY instruction at &8CF8

8CFB:iny

Decrement the remaining length of the string

8CFC:dex

Continue until the remaining length becomes zero

8CFD:bne &8CF5

Place a carriage return at the end of the string

8CFF:lda #&D

8D01:bne &8CE9

This section deals with numeric entries. The first step is to zero Y to point to the first byte on the stack

8D03:ldy #&0

Get the first byte

8D05:lda (&4),Y

Save it

8D07:sta (&37),Y

Increment Y to point to the next byte

8D09:iny

Compare Y with the type of the item. If Y (which is 1) is less than or equal to the type, we are dealing with an eight bit byte and it has been transferred correctly, so we can exit this routine

8D0A:cpy &39

8D0C:bcs &8D26

Transfer the second byte

8D0E:lda (&4),Y

8D10:sta (&37),Y**8D12:iny**

Transfer the third byte

8D13:lda (&4),Y**8D15:sta (&37),Y****8D17:iny**

Transfer the fourth byte

8D18:lda (&4),Y**8D1A:sta (&37),Y**

Exit if the type of the item was integer

8D1C:iny**8D1D:cpy &39****8D1F:bcs &8D26**

Transfer the fifth and final byte

8D21:lda (&4),Y**8D23:sta (&37),Y****8D25:iny**

Get the length of the item into A

8D26:tya

Add this value onto the stack pointer to remove the value from the BASIC stack

8D27:clc**8D28:jmp &BDE1****PRINT # statement routine**

This routine does not appear in the action address tables, since it is called from the PRINT routine when a hash character is encountered

Back PTR # 1 over the hash character

8D2B:dec &A

Work out the file handle

8D2D:jsr &BFA9

Copy the handle into A

8D30:tya

Save it on the system stack

8D31:pha

Get the next character

8D32:jsr &8A8C

Exit if it is not a comma

8D35:cmp #&2C**8D37:bne &8D77**

Evaluate the expression

8D39:jsr &9B29

Pack FAC # 1 to &46C onwards; this is done because the packed form of floating point numbers is saved

8D3C:jsr &A385

Get the file handle

8D3F:pla
Transfer it into Y
8D40:tay
Get the type of the item
8D41:lda &27
BPUT the type of the item to the file
8D43:jsr &FFD4
Get the type of the item into X
8D46:tax
Goto &8D64 if the item is a string
8D47:beq &8D64
Goto &8D57 if the item is real
8D49:bmi &8D57
Point to the last byte of the IAC
8D4B:ldx #&3
Get the next byte from the IAC
8D4D:lda &2A,X
Send it out
8D4F:jsr &FFD4
Point to the next byte
8D52:dex
Continue until all the bytes have been output
8D53:bpl &8D4D
Go back to check for a comma
8D55:bmi &8D30
Set X to the length of a real number
8D57:ldx #&4
Get the next byte of the real number
8D59:lda &46C,X
Output it
8D5C:jsr &FFD4
Continue the process until the entire number has been output
8D5F:dex
8D60:bpl &8D59
Go back to check for a comma
8D62:bmi &8D30
Get the length of the string
8D64:lda &36
Output it
8D66:jsr &FFD4
Get the length into X
8D69:tax
Go back to check for a comma if the length is zero
8D6A:beq &8D30

Get the next byte of the string

8D6C:lda &5FF,X

Output it

8D6F:jsr &FFD4

Point to the next byte

8D72:dex

Continue until the end of the string

8D73:bne &8D6C

Go back to look for a comma

8D75:beq &8D30

Get the handle back from the stack to avoid corrupting it

8D77:pla

Save the current offset

8D78:sty &A

Exit

8D7A:jmp &8B98

This routine is entered when the end of the PRINT statement is encountered. It first moves the cursor to a new line and then exits

8D7D:jsr &BC25

Exit

8D80:jmp &8B96

This routine is entered when a semi-colon is encountered in the PRINT statement. It first zeros the field width byte and the hex flag

8D83:lda #&0

8D85:sta &14

8D87:sta &15

Get the next character

8D89:jsr &8A97

Exit (without moving to a new line) if it is a colon

8D8C:cmp #&3A

8D8E:beq &8D80

Exit (without moving to a new line) if it is a carriage return

8D90:cmp #&D

8D92:beq &8D80

Exit (without moving to a new line) if it is the word ELSE

8D94:cmp #&8B

8D96:beq &8D80

Continue scanning the item

8D98:bne &8DD2

PRINT statement routine

Get the character following the word PRINT

8D9A:jsr &8A97

Enter the 'PRINT #' routine if the character was a '#'

8D9D:cmp #&23

8D9F:beq &8D2B

Decrement the pointer to account for not finding the hash character

8DA1:dec &A

Join the middle of the PRINT loop

8DA3:jmp &8DBB

This routine is called when a comma is encountered

Get the current field width

8DA6:lda &400

Leave the comma routine if the field width is zero

8DA9:beq &8DBB

Get COUNT

8DAB:lda &1E

Exit if it is zero - thus, 'PRINT ',,,' "HELLO"' does not work

8DAD:beq &8DBB

Subtract the field width from COUNT

8DAF:sub &400

If there was no overflow, go back to do it again

8DB2:bcs &8DAD

Transfer the number of spaces to be printed into Y. The number is held as a negative, twos complement number

8DB4:tax

Print a space

8DB5:jsr &B565

Continue printing until enough spaces have been printed

8DB8:iny

8DB9:bne &8DB5

This is the main PRINT loop

Copy the field width into zero page

8DBB:clc

8DBC:lda &400

8DBF:sta &14

Set the hex flag to the status of the carry flag. On entry at &8DBB, the hex flag is thus cleared

8DC1:ror &15

Get the next character

8DC3:jsr &8A97

Join the colon routine if the character was a colon

8DC6:cmp #&3A

8DC8:beq &8D7D

Join the colon routine if the character was a carriage return

8DCA:cmp #&D

8DCC:beq &8D7D

Join the colon routine if the character was the word ELSE

8DCE:cmp #&8B

8DD0:beq &8D7D

If the character was '∼', the carry flag will be set, so this branch will rather cleverly set the hex flag

8DD2:cmp #&7E

8DD4:beq &8DC1

Join the comma routine if the character was a comma

8DD6:cmp #&2C

8DD8:beq &8DA6

Join the semi-colon routine if the character was a semi-colon

8DDA:cmp #&3B

8DDC:beq &8D83

Check for some of the items that are common to both INPUT and PRINT

8DDE:jsr &8E70

Go back to get another character (without changing the hex flag) if an item was found

8DE1:bcc &8DC3

Save the hex flag and the current field width. This is so that a line like 'PRINT ∼FN_aFN_b' will work if both functions include decimal 'PRINT' statements

8DE3:lda &14

8DE5:pha

8DE6:lda &15

8DE8:pha

Decrement the offset of PTR #2 to account for not finding anything so far

8DE9:dec &1B

Evaluate the expression

8DEB:jsr &9B29

Restore the field width and the hex flag

8DEE:pla

8DEF:sta &15

8DF1:pla

8DF2:sta &14

Copy the offset of PTR #2 to PTR #1

8DF4:lda &1B

8DF6:sta &A

Get the type of the expression

8DF8:tya

Goto &8E0E if the expression is a string

8DF9:beq &8E0E

Convert the number to a string

8DFB:jsr &9EDF

Subtract the length of the number from the current field width

8DFE:lda &14

8E00:sec

8E01:sbc &36

Go straight to print the number if it was longer or the same length as the field width

8E03:bcc &8E0E

8E05:beq &8E0E

Transfer the number of leading spaces to print into Y

8E07:tay

Print a space

8E08:jsr &B565

Decrement the space count

8E0B:dey

Continue until enough spaces have been printed

8E0C:bne &8E08

This routine prints strings

Get the length of the string

8E0E:lda &36

Exit if the string was null

8E10:beq &8DC3

Point to the first character of the string

8E12:ldy #&0

Get the current character of the string

8E14:lda &600,Y

Print it

8E17:jsr &B558

Point to the next character

8E1A:iny

Check for the end of the string

8E1B:cpy &36

Continue if we have not reached the end

8E1D:bne &8E14

Otherwise, go back to join the main loop

8E1F:beq &8DC3

Direct jump to the 'Missing ,' error

8E21:jmp &8AA2

TAB(X,Y) routine

This routine is only entered via the TAB(X) routine. On entry, the IAC contains the X coordinate of the destination cursor position

Give a 'Missing ,' error if the next character is not a comma

8E24:cmp #&2C

8E26:bne &8E21

Save the X coordinate of the destination

8E28:lda &2A

8E2A:pha

Evaluate the next expression, and check for the right hand bracket

8E2B:jsr &AE56

Ensure the result was an integer

8E2E:jsr &92F0

Do a VDU 31

8E31:lda #&1F

8E33:jsr &FFEE

Get the X coordinate back

8E36:pla

Send it to the VDU driver

8E37:jsr &FFEE

Print the Y coordinate

8E3A:jsr &9456

Exit

8E3D:jmp &8E6A

TAB(X) routine

Get the integer expression

8E40:jsr &92DD

Get the next character

8E43:jsr &8A8C

Try doing a 'TAB(X,Y)' if it is not a right hand bracket

8E46:cmp #&29

8E48:bne &8E24

Get the destination column

8E4A:lda &2A

Subtract it from the current column

8E4C:sbc &1E

Exit if we are already there

8E4E:beq &8E6A

Save the difference in Y

8E50:tay

If the difference is positive, skip to &8E5F

8E51:bcs &8E5F

Move to a new line

8E53:jsr &BC25

Skip getting the integer expression in the SPC routine, then print the number of spaces that were the argument to the 'TAB' item

8E56:beq &8E5B

SPC routine

Evaluate the argument

8E58:jsr &92E3

Get the number of spaces to print

8E5B:ldy &2A

Exit if the argument is zero

8E5D:beq &8E6A

Print a space

8E5F:jsr &B565

Continue printing another one if necessary

8E62:dey

8E63:bne &8E5F

Else exit

8E65:beq &8E6A

New line routine

Move to a new line

8E67:jsr &BC25

Clear the carry flag to indicate that an item was found and dealt with

8E6A:clc

Make PTR #1 the same as PTR #2

8E6B:ldy &1B

8E6D:sty &A

Exit

8E6F:rts

Get the items common to PRINT and INPUT

Set PTR #2 to PTR #1

8E70:ldx &B

8E72:stx &19

8E74:ldx &C

8E76:stx &1A

8E78:ldx &A

8E7A:stx &1B

Check for a single quote

8E7C:cmp #&27

8E7E:beq &8E67

Check for 'TAB('

8E80:cmp #&8A

8E82:beq &8E40

Check for 'SPC'

8E84:cmp #&89

8E86:beq &8E58

Set the carry flag to indicate that nothing was found

8E88:sec

Exit

8E89:rts

Get INPUT items

This routine handles all the parts of the INPUT statement except variable names.

Get the next character

8E8A:jsr &8A97

Check for the PRINT/INPUT items

8E8D:jsr &8E70

Exit if one or more was found

8E90:bcc &8E89

Go to the quotes routine if the character was a double quote

8E92:cmp #&22**8E94:beq &8EA7**

Set the carry flag to indicate failure

8E96:sec

Exit

8E97:rts**8E98:brk****8E99:09-****8E9A:4D-M****8E9B:69-i****8E9C:73-s****8E9D:73-s****8E9E:69-i****8E9F:6E-n****8EA0:67-g****8EA1:20-****8EA2:22-"****8EA3:00-**

Print the character

8EA4:jsr &B558**Routine to print a quoted string**

This routine is only used by the INPUT statement

Get the character after the quote

8EA7:iny**8EA8:lda (&19),Y**

Give a 'Missing "' error if it is a carriage return

8EAA:cmp #&D**8EAC:beq &8E98**

Go back if it is not another quote

8EAE:cmp #&22**8EB0:bne &8EA4**

Get the next character

8EB2:iny**8EB3:sty &1B****8EB5:lda (&19),Y**

If it is another quote, print it, and continue looking for the closing quote (this deals with 'INPUT "HELLO ""THERE" G\$'). If it is not another quote, exit

8EB7:cmp #&22**8EB9:bne &8E6A**

8EBB:beq &8EA4

CLG statement routine

Check for the end of the statement

8EBD:jsr &9857

Load the code for CLG

8EC0:lda #&10

Jump into the CLS code

8EC2:bne &8ECC

CLS statement routine

Check for the end of the statement

8EC4:jsr &9857

Zero COUNT

8EC7:jsr &BC28

Get the code for CLS

8ECA:lda #&C

Output the code

8ECC:jsr &FFEE

Exit

8ECF:jmp &8B9B

CALL statement routine

Evaluate the address of the routine

8ED2:jsr &9B1D

Ensure it is an integer

8ED5:jsr &92EE

Save the call address on the stack

8ED8:jsr &BD94

Zero the the number of parameters

8EDB:ldy #&0

8EDD:sty &600

Create a pointer into the parameter area

8EE0:sty &6FF

Get the next character

8EE3:jsr &8A8C

Skip forwards if it is not a comma

8EE6:cmp #&2C

8EE8:bne &8F0C

Get a variable name

8EEA:ldy &1B

8EEC:jsr &95D5

Give a 'No such variable' error if the variable is invalid or does not exist

8EEF:beq &8F1B

Get the pointer into the parameter area

8EF1:ldy &6FF

Increment the pointer to point to the LSB of the parameter address

8EF4:iny

Get the LSB of the parameter address

8EF5:lda &2A

Save it

8EF7:sta &600,Y

Store the MSB of the parameter address

8EFA:iny

8EFB:lda &2B

8EFD:sta &600,Y

Store the type of the parameter

8F00:iny

8F01:lda &2C

8F03:sta &600,Y

Increment the number of parameters

8F06:inc &600

Go back to search for another parameter

8F09:jmp &8EE0

Decrement the pointer over the character that was not a comma

8F0C:dec &1B

Check for the end of the statement

8F0E:jsr &9852

Pull the address off the stack

8F11:jsr &BDEA

Call the address

8F14:jsr &8F1E

Make sure we are not in decimal mode

8F17:cld

Exit

8F18:jmp &8B9B

Direct jump to 'No such variable' error message

8F1B:jmp &AE43

Call a user routine

This routine is used by CALL and USR to call a machine code routine whose address is in the IAC. In addition, it copies the variable A%, X% and Y% to the relevant registers, and copies the least significant bit of the C% variable into the carry flag.

Get the least significant byte of C%

8F1E:lda &40C

Get the least significant bit into the carry flag

8F21:lsr A

Get the accumulator from the LSB of A%

8F22:lda &404

Get X from the LSB of X%

8F25:ldx &460

Get Y from the LSB of Y%

8F28:ldy &464

Execute the routine. The RTS at the end of the routine will pass control back to the point where this subroutine was called

8F2B:jmp (&2A)

Direct jump to 'Syntax error'

8F2E:jmp &982A

DELETE command routine

Decode the line number after the word DELETE

8F31:jsr &97DF

Give a 'Syntax error' if no line number is found

8F34:bcc &8F2E

Push the line number on the stack

8F36:jsr &BD94

Get the next character

8F39:jsr &8A97

Give a 'Syntax error' if it is not a comma

8F3C:cmp #&2C

8F3E:bne &8F2E

Decode the next line number

8F40:jsr &97DF

Give a 'Syntax error' if one is not found

8F43:bcc &8F2E

Check for the end of the command

8F45:jsr &9857

Copy the second line number to (&39)

8F48:lda &2A

8F4A:sta &39

8F4C:lda &2B

8F4E:sta &3A

Get back the starting line number

8F50:jsr &BDEA

Delete the line whose number is in the IAC

8F53:jsr &BC2D

Check the escape key

8F56:jsr &987B

Increment the IAC

8F59:jsr &9222

Continue if the line number in the IAC is not greater than that stored in &39,&3A

8F5C:lda &39

8F5E:cmp &2A

8F60:lda &3A

8F62:sbc &2B**8F64:bcs &8F53**

Do a warm start

8F66:jmp &8AF3**Decode parameters for AUTO and RENUMBER**

This subroutine is used by both RENUMBER and AUTO to decode the parameters of the commands and to carry out some initialisation that is common to them both. It leaves the starting line number on the stack and the interval in the IAC. PAGE + 1 is copied to &37,&38 and TOP is copied to &3B,&3C

Load the IAC with 10, the default first parameter

8F69:lda #&A**8F6B:jsr &AED8**

Decode the first line number

8F6E:jsr &97DF

Push the line number on the stack. If no line number was found, the default of 10 will be pushed

8F71:jsr &BD94

Load the IAC with 10, the default second parameter

8F74:lda #&A**8F76:jsr &AED8**

Get the next character

8F79:jsr &8A97

Jump forwards to skip getting the second parameter if the character was not a comma

8F7C:cmp #&2C**8F7E:bne &8F8D**

Decode the second line number

8F80:jsr &97DF

Give a 'Silly' error message if the interval is greater than 255

8F83:lda &2B**8F85:bne &8FDF**

Give a 'Silly' error message if the interval is zero

8F87:lda &2A**8F89:beq &8FDF**

Adjust the offset of PTR #1 to allow for the branch at &8F7E

8F8B:inc &A**8F8D:dec &A**

Check for the end of the command

8F8F:jmp &9857

Copy TOP to &3B,&3C

8F92:lda &12**8F94:sta &3B****8F96:lda &13****8F98:sta &3C**

Copy PAGE + 1 to &37,&38

8F9A:lda &18

8F9C:sta &38

8F9E:lda #&1

8FA0:sta &37

Exit

8FA2:rts

RENUMBER command routine

The method used is to copy all the line numbers in the program, in sequence, to a pile from TOP onwards.

Then all the line numbers at the start of each line are renumbered. Finally, the program is scanned for line number tokens. When one is found, the position of the line number is found in the list of line numbers above the program. From this position, the new line number is found in the program and substituted

Decode the parameters

8FA3:jsr &8F69

Pull the starting line number to &39 - &3C

8FA6:ldx #&39

8FA8:jsr &BE0D

Check for 'Bad program'

8FAB:jsr &BE6F

Copy TOP to &3B,&3C again - it was overwritten when we pulled the starting line number

8FAE:jsr &8F92

Get the MSB of the current line number in the program

8FB1:ldy #&0

8FB3:lda (&37),Y

Jump forwards if we have met the end of the program

8FB5:bmi &8FE7

Save the MSB of the number in the pile at TOP

8FB7:sta (&3B),Y

Copy the LSB in the same way

8FB9:iny

8FBA:lda (&37),Y

8FBC:sta (&3B),Y

Add two to the pointer at (&3B). This moves the pointer where the line numbers are stored

8FBE:sec

8FBF:tya

8FC0:adc &3B

8FC2:sta &3B

8FC4:tax

8FC5:lda &3C

8FC7:adc #&0**8FC9:sta &3C**

Give a 'RENUMBER space' error message if the pointer colides with HIMEM

8FCB:cpx &6**8FCD:sbc &7****8FCF:bcs &8FD6**

Move the line number pointer at &37,&38 to the next line

8FD1:jsr &909F

Loop back to get the next line

8FD4:bcc &8FB1

Give the 'RENUMBER space' error message

8FD6:brk**8FD7:00-****8FD8:CC-L****8FD9:20-****8FDA:73-s****8FDB:70-p****8FDC:61-a****8FDD:63-c****8FDE:65-e**

Give the 'Silly' error message

8FDF:brk**8FE0:00-****8FE1:53-S****8FE2:69-i****8FE3:6C-l****8FE4:6C-l****8FE5:79-y****8FE6:00-**

Copy PAGE + 1 to &37,&38

8FE7:jsr &8F9A

Get the MSB of the current line number in the program

8FEA:ldy #&0**8FEC:lda (&37),Y**

Exit if the end of the program has been reached

8FEE:bmi &900D

Update the line number of the program line currently pointed to

8FF0:lda &3A**8FF2:sta (&37),Y****8FF4:lda &39****8FF6:iny****8FF7:sta (&37),Y**

Add the interval to the current line number to give the next line number to be used

8FF9:clc**8FFA:lda &2A**

```

8FFC:adc &39
8FFE:sta &39
9000:lda #&0
9002:adc &3A

```

(This instruction makes sure that the line number does not exceed 32767)

```

9004:and #&7F
9006:sta &3A

```

Move the pointer at (&37) to the next line

```
9008:jsr &909F
```

Go back for the next line

```
900B:bcc &8FEA
```

Copy PAGE to PTR#1

```

900D:lda &18
900F:sta &C
9011:ldy #&0
9013:sty &B

```

Get the MSB of the line number of the current line

```

9015:iny
9016:lda (&B),Y

```

Exit if the end of the program has been reached

```
9018:bmi &903A
```

Point to the first byte of text of the line

```
901A:ldy #&4
```

Get a byte from the text

```
901C:lda (&B),Y
```

Skip to &903D if a line number token was found

```

901E:cmp #&8D
9020:beq &903D

```

Point to the next byte

```
9022:iny
```

Continue the search until a carriage return is found

```

9023:cmp #&D
9025:bne &901C

```

Exit if the end of the program has been reached

```

9027:lda (&B),Y
9029:bmi &903A

```

Get the length of the current line

```

902B:ldy #&3
902D:lda (&B),Y

```

Add this to PTR#1, to point to the start of the next line, and then go backwards to continue the search

```

902F:clc
9030:adc &B
9032:sta &B
9034:bcc &901A

```


9036:inc &C**9038:bcx &901A**

Do a warm start when the search has finished

903A:jmp &8AF3

Decode the line number

903D:jsr &97EB

Copy TOP to &3B,&3C and PAGE + 1 to &37,&38

9040:jsr &8F92

Get the MSB of the line number of the current line in the program

9043:ldy #&0**9045:lda (&37),Y**

Exit if the end of the program has been reached

9047:bmi &9080

Get the MSB of the next line number in the pile

9049:lda (&3B),Y

Point to the LSB of the same line number

904B:iny

Check it against the line number being sought

904C:cmp &2B

Skip forwards if they do not match

904E:bne &9071

Get the LSB of the line number from the pile

9050:lda (&3B),Y

Compare it with the LSB of the line number being sought

9052:cmp &2A

Go to &9071 if they do not match

9054:bne &9071

Get the renumbered equivalent of the line number in the pile, and store it in &3D,&3E

9056:lda (&37),Y**9058:sta &3D****905A:dey****905B:lda (&37),Y****905D:sta &3E**

Get the offset of the line number token from PTR # 1

905F:ldy &A**9061:dey**

Copy PTR # 1 to &37,&38

9062:lda &B**9064:sta &37****9066:lda &C****9068:sta &38**

Encode the line number and store it

906A:jsr &88F5

Go back to continue the search

906D:ldy &A
906F:bne &901C

Move &37,&38 to the next line

9071:jsr &909F

Add two to &3B,&3C and go back to look for the line number

9074:lda &3B
9076:adc #&2
9078:sta &3B
907A:bcc &9043
907C:inc &3C
907E:bcs &9043

This routine is used to print the 'Failed at XXXX' message when a reference to a non-existent line number is found. The problem is that it uses the routine at &BFCF, which prints the text stored in memory following the JSR &BFCF instruction up to the first character with bit 7 set. This is not disassembled correctly and appears as follows:

9080:jsr &BFCF
9083:lsr &61
9085:adc #&6C
9087:adc &64
9089:jsr &7461
908C:jsr &B1C8
908F:0B-
9090:85-
9091:2B- +
9092:C8-H
9093:B1-1
9094:0B-
9095:85-
9096:2A-*
9097:20-
9098:1F-
9099:99-
909A:20-
909B:25-%
909C:BC-<
909D:F0-p
909E:CE-N

It should actually appear as follows:

Print the message

9080:jsr &BFCF
9081:"Failed at "

Get the line number that couldn't be found into the IAC

```

908D:iny
908E:lda (&B),Y
9090:sta &2B
9092:iny
9093:lda (&B),Y
9095:sta &2A

```

Print the IAC

```
9097:jsr &991F
```

Move to a new line

```
909A:jsr &BC25
```

Join the original code

```
909D:beq &906D
```

Point to next line

This routine makes &37,&38 point to the start of the next line.

Make Y point to the byte giving the length of line

```
909F:iny
```

Get the length of the line

```
90A0:lda (&37),Y
```

Add it to the pointer

```
90A2:adc &37
```

```
90A4:sta &37
```

```
90A6:bcc &90AB
```

```
90A8:inc &38
```

Make sure that the carry flag is clear when the routine finishes, since this allows BCC to be used as an unconditional branch instruction. This is demonstrated at &8FD4

```
90AA:clc
```

Exit

```
90AB:rts
```

AUTO command routine

Remember that the AUTO routine is terminated by pressing 'Escape', and so does not contain any explicit means of exiting (except by making a line number over 32767 appear)

Decode the parameters to the command

```
90AC:jsr &8F69
```

Save the interval on the machine stack

```
90AF:lda &2A
```

```
90B1:pha
```

Get the starting line number back

```
90B2:jsr &BDEA
```

Save it on the machine stack again

```
90B5:jsr &BD94
```

Print the IAC using a field width of five characters

90B8:jsr &9923

Get a line of text, using a space as the prompt

90BB:lda #&20

90BD:jsr &BC02

Pull the line number back

90C0:jsr &BDEA

Tokenise the line

90C3:jsr &8951

Insert the line into the program

90C6:jsr &BC8D

Clear the variable area and all the stacks

90C9:jsr &BD20

Get the interval back

90CC:pla

90CD:pha

Add the interval to the current line number and return for the next line

90CE:clc

90CF:adc &2A

90D1:sta &2A

90D3:bcc &90B5

90D5:inc &2B

90D7:bpl &90B5

Exit via a warm start if the line number becomes over 32767

90D9:jmp &8AF3

Direct jump to 'DIM space' error message

90DC:jmp &9218

Reserve space DIM

This routine is used when DIM is used to reserve space, as in 'DIM P% 200'. The action of the routine is to search for and, if necessary, create the variable indicated, then set it to VARTOP. VARTOP is then raised by the indicated amount. The routine also checks the amount of available memory

This routine is entered via the main DIM routine when the opening bracket of the array name is missing. Thus, PTR#1 has to be moved back to allow for the opening bracket not being found

90DF:dec &A

Get the name of the variable

90E1:jsr &9582

Give a 'Bad DIM' error message if the variable is a string or is invalid

90E4:beq &9127

90E6:bcx &9127

Save the address and type of the variable

90E8:jsr &BD94

Get the integer expression giving the number of bytes to reserve

90EB:jsr &92DD

Increment the IAC to allow for the zeroth element of the byte array

90EE:jsr &9222

Give a 'Bad DIM' error message if an attempt is made to reserve more than 65535 bytes (a useful feature on a machine with 32K RAM)

90F1:lda &2D

90F3:ora &2C

90F5:bne &9127

Add the LSB of the size reserved to the LSB of VARTOP

90F7:clc

90F8:lda &2A

90FA:adc &2

Place the answer in Y

90FC:tay

Add the MSB of the size reserved to the MSB of VARTOP

90FD:lda &2B

90FF:adc &3

Place the answer in X. Thus, X(msb) and Y(lsb) contain the value VARTOP would take on if the DIM statement were carried out

9101:tax

Compare this value with the top of the stack

9102:cpy &4

9104:sbx &5

Give a 'DIM space' message if there is not enough space

9106:bcs &90DC

Save the current VARTOP value in the IAC. This is the first address of the space reserved

9108:lda &2

910A:sta &2A

910C:lda &3

910E:sta &2B

Save the new value for VARTOP

9110:sty &2

9112:stx &3

Zero the top two bytes of the old VARTOP value in the IAC

9114:lda #&0

9116:sta &2C

9118:sta &2D

Indicate that an integer value is to be assigned to the variable

911A:lda #&40

911C:sta &27

Assign the variable

911E:jsr &B4B4

Update PTR # 1 from the offset of PTR # 2

9121:jsr &8827

Join the main DIM code

9124: jmp &920B

Give a 'Bad DIM' error message

9127: brk

9128: 0A-

9129: 42-B

912A: 61-a

912B: 64-d

912C: 20-

912D: DE-^

912E: 00-

DIM statement routine

Skip any spaces after the word DIM

912F: jsr &8A97

Get the offset from PTR # 1 into A

9132: tya

Add it to PTR # 1, leaving the result in A(lsb) and X(msb)

9133: clc

9134: adc &B

9136: ldx &C

9138: bcc &913C

913A: inx

913B: clc

Save PTR # 1-1 in (&37)

913C: sbc #&0

913E: sta &37

9140: txa

9141: sbc #&0

9143: sta &38

Location &3F is used to hold the number of bytes required by each element of the current array. This is initialised to 5 bytes, which is correct for real arrays. Integer and string arrays decrement this value

9145: ldx #&5

9147: stx &3F

Get the name of the array being defined. The normal variable name routine cannot be used because of the way it allows for the indirection operators

9149: ldx &A

914B: jsr &9559

Give a 'Bad DIM' error message if no name was found. The system can tell that no name was found because Y points, on exit, to the letter that couldn't be fitted into the name. If it still points to the first character on exit, no name was found

914E: cpy #&1

9150: beq &9127

Jump forwards if the character that couldn't be coped with was an opening bracket. If

this occurs, the array in question is a real array, and so location &3F does not need to be adjusted

9152:cmp #&28

9154:beq &916B

Jump forwards if the character is a '\$'

9156:cmp #&24

9158:beq &915E

Jump forwards if the character is not a '%'

915A:cmp #&25

915C:bne &9168

Decrement the number of bytes in each element to correspond to integer and string arrays

915E:dec &3F

Point to the next character after the '%' or the '\$'

9160:iny

9161:inx

Get the character

9162:lda (&37),Y

Jump forwards if the character is an opening bracket. If the character is not a '(' the routine to handle reserve space DIMs is called

9164:cmp #&28

9166:beq &916B

Call the reserve space DIM routine

9168:jmp &90DF

Save the two offsets. Y contains the length of the name of the array and X points to the character after the name, offset from PTR #1

916B:sty &39

916D:stx &A

Find the address of the variable name

916F:jsr &9469

Give a 'Bad DIM' error if an address was found for the name, since this would indicate that the array is being redimensioned

9172:bne &9127

Create a catalogue entry for the array name

9174:jsr &94FC

Clear the single byte after the name, and make VARTOP point to the byte after the zero byte terminating the name

9177:ldx #&1

9179:jsr &9531

Save the number of bytes in each element on the machine stack

917C:lda &3F

917E:pha

Save the current offset into the subscript area on the machine stack

917F:lda #&1

9181:pha

Set the IAC to 1

9182:jsr &AED8

Save the IAC on the BASIC stack

9185:jsr &BD94

Get the expression giving the next dimension

9188:jsr &8821

Give a 'Bad DIM' error message if the subscript is greater than 16383

918B:lda &2B

918D:and #&C0

918F:ora &2C

9191:ora &2D

9193:bne &9127

Increment the IAC, to allow for element zero

9195:jsr &9222

Get the offset into the subscript area

9198:pla

Move it into Y so that it can be used as an index

9199:tay

Get the LSB of the subscript

919A:lda &2A

Put it in the subscript area

919C:sta (&2),Y

Point to the MSB of the subscript

919E:iny

Get the MSB of the subscript

919F:lda &2B

Store it

91A1:sta (&2),Y

Point to the LSB of the next subscript

91A3:iny

Save the pointer on the stack

91A4:tya

91A5:pha

Pull the IAC from the stack, and multiply it by the current subscript. The idea is that the IAC will then contain the number of elements so far in the array

91A6:jsr &9231

Get the next character

91A9:jsr &8A97

Go back for the next subscript if it is a comma

91AC:cmp #&2C

91AE:beq &9185

Skip the error if it is a closing bracket

91B0:cmp #&29

91B2:beq &91B7

Give a 'Bad DIM' error

91B4: jmp &9127

Get and store the offset into the subscript storage area

91B7: pla**91B8: sta &15**

Get and store the number of bytes in each element

91BA: pla**91BB: sta &3F**

Zero location &40

91BD: lda #&0**91BF: sta &40**

Multiply locations &3F,&40 by the IAC and leave the result in the IAC. The IAC now contains the total number of bytes required by the array

91C1: jsr &9236

Store the subscript area offset immediately after the zero following the name of the array

91C4: ldy #&0**91C6: lda &15****91C8: sta (&2),Y**

Add this offset to the number of bytes taken up by the elements of the array. The IAC now contains the total number of bytes required by the array, including the subscript data

91CA: adc &2A**91CC: sta &2A****91CE: bcc &91D2****91D0: inc &2B**

Copy VARTOP to locations &37,&38

91D2: lda &3**91D4: sta &38****91D6: lda &2****91D8: sta &37**

Add VARTOP to the length of the array

91DA: clc**91DB: adc &2A****91DD: tay****91DE: lda &2B****91E0: adc &3**

Give the error message 'DIM space' if the total is greater than 65535

91E2: bcs &9218

Check the proposed new VARTOP against the bottom of the stack, giving a 'DIM space' error message if there is not enough room

91E4: tax**91E5: cpy &4****91E7: sbc &5****91E9: bcs &9218**

Save the new VARTOP value

91EB:sty &2

91ED:stx &3

The rest of the code is concerned with clearing the area just dimensioned. The first step is to make a more efficient pointer out of locations &37,&38. Location &37 is cleared, and Y is set to the offset from the two locations of the first element of the array

91EF:lda &37

91F1:adc &15

91F3:tay

91F4:lda #&0

91F6:sta &37

91F8:bcc &91FC

91FA:inc &38

Store zero at the current location in the array

91FC:sta (&37),Y

Increment the LSB of the pointer

91FE:iny

Do not increment the MSB unless it is necessary

91FF:bne &9203

9201:inc &38

Check the LSB of the pointer against the LSB of VARTOP

9203:cpy &2

If they are not the same, go back to clear the next location

9205:bne &91FC

Check the MSB of the pointer against the MSB of VARTOP

9207:cpx &38

Continue if they are not the same

9209:bne &91FC

Get the next character

920B:jsr &8A97

Go back for the next array if the character is a comma

920E:cmp #&2C

9210:beq &9215

Exit

9212:jmp &8B96

Go back and get the next array

9215:jmp &912F

'DIM space' error message

9218:brk

9219:0B-

921A:DE-^

921B:20-

921C:73-s

921D:70-p

921E:61-a

921F:63-c

9220:65-e**9221:00-****Increment the IAC**

Notice that no check for overflow is made

Increment the LSB, and exit if it does not become zero

9222:inc &2A**9224:bne &9230**

Increment the next byte, and exit if it does not become zero

9226:inc &2B**9228:bne &9230**

Increment the next byte, and exit if it does not become zero

922A:inc &2C**922C:bne &9230**

Increment the MSB

922E:inc &2D

Exit

9230:rts**Pull IAC to &3F,&40,&41,&42**

Set the destination for the pull operation to &3F

9231:ldx #&3F

Pull four bytes off the stack to X, X + 1, X + 2, X + 3

9233:jsr &BE0D

Fall into multiplication routine

Set &2A,&2B = (&2A,&2B)*(&3F,&40)

This routine is used by DIM to work out the amount of memory required for a given array. It uses a similar method to that outlined earlier in this book. Notice carefully the way X and Y are used to minimize memory accesses. You cannot use this routine directly in assembly language programs, since it gives a 'Bad DIM' error if overflow occurs. There is, of course, nothing stopping you deriving a routine similar to it

X(msb) and Y(lsb) are used as the accumulator where the answer is built up. They must be cleared on entry. If they are not, their contents are added to the answer

9236:ldx #&0**9238:ldy #&0**

Get the least significant bit of the multiplicand

923A:lsr &40**923C:ror &3F**

Do not add the multiplier to the accumulator if the bit is clear

923E:bcc &924B

Add the multiplier to the accumulator

9240:clc**9241:tya**

9242:adc &2A

9244:tay

9245:txa

9246:adc &2B

9248:tax

Give a 'Bad DIM' error message if overflow is encountered

9249:bcs &925A

Multiply the multiplier by two

924B:asl &2A

924D:rol &2B

Continue this process until the multiplicand becomes zero

924F:lda &3F

9251:ora &40

9253:bne &923A

Save the accumulator as the answer

9255:sty &2A

9257:stx &2B

Exit

9259:rts

Direct jump to 'Bad DIM' error message

925A:jmp &9127

HIMEM statement routine

This routine copes with the HIMEM statement - in other words, it deals with assignments to HIMEM, but does not deal with HIMEM when it appears in expressions

Check for an equals sign, and evaluate the integer expression following it

925D:jsr &92EB

Set the stack pointer and HIMEM to this address

9260:lda &2A

9262:sta &6

9264:sta &4

9266:lda &2B

9268:sta &7

926A:sta &5

Exit

926C:jmp &8B9B

LOMEM statement routine

This routine deals with assignments to LOMEM, not with the cases when LOMEM appears in expressions

Check for an equals sign, and evaluate the integer expression following it

926F:jsr &92EB

Use this address to set LOMEM and VARTOP

9272:lda &2A


```

9274:sta &0
9276:sta &2
9278:lda &2B
927A:sta &1
927C:sta &3

```

Clear the variable catalogue

```
927E:jsr &BD2F
```

Exit

```
9281:beq &928A
```

PAGE statement routine

This routine copes with assigning a new value to PAGE, not with interrogating the present value of PAGE

Check for an equals sign, and evaluate the integer expression following it

```
9283:jsr &92EB
```

Use this address to set PAGE

```
9286:lda &2B
```

```
9288:sta &18
```

Exit

```
928A:jmp &8B9B
```

CLEAR statement routine

Check for the end of the statement

```
928D:jsr &9857
```

Clear the variable catalogue

```
9290:jsr &BD20
```

Exit

```
9293:beq &928A
```

TRACE statement routine

The word TRACE can be followed by 'ON', 'OFF' or a line number. If 'TRACE ON' is used, the TRACE flag is set and the TRACE line number is set to &FFXX. If 'TRACE OFF' is used, the TRACE flag is reset. IF 'TRACE X' is used, the TRACE flag is set and the TRACE line number is set to X

Check for a line number after TRACE

```
9295:jsr &97DF
```

Skip forwards if a line number was present

```
9298:bcs &92A5
```

Check for the word 'ON'

```
929A:cmp #&EE
```

```
929C:beq &92B7
```

Check for the word 'OFF'

```
929E:cmp #&87
```

```
92A0:beq &92C0
```

Evaluate the integer expression following TRACE

92A2:jsr &8821

Check that nothing follows the argument to TRACE

92A5:jsr &9857

Save the line number as the TRACE line number

92A8:lda &2A

92AA:sta &21

92AC:lda &2B

92AE:sta &22

Indicate that TRACE is ON

92B0:lda #&FF

Save the TRACE ON/OFF flag

92B2:sta &20

Exit

92B4:jmp &8B9B

Increment the pointer past the word 'ON'

92B7:inc &A

Check that nothing follows the word 'ON'

92B9:jsr &9857

Make the MSB of the TRACE line number &FF. This will ensure that all line numbers are less than the TRACE number, which in turn means that all lines will be printed

92BC:lda #&FF

92BE:bne &92AE

Increment the pointer past the word 'OFF'

92C0:inc &A

Check that nothing follows the word 'OFF'

92C2:jsr &9857

Set the TRACE flag to off

92C5:lda #&0

92C7:beq &92B2

TIME statement routine

This routine deals with assignments to TIME, rather than occurrences of TIME in expressions

Check for an equals sign and evaluate the integer expression following it

92C9:jsr &92EB

Make X and Y point to the IAC in OSWORD format

92CC:ldx #&2A

92CE:ldy #&0

Make the fifth parameter of the call be zero

92D0:sty &2E

Indicate that we wish to set the clock

92D2:lda #&2

Call OSWORD

92D4:jsr &FFF1

Exit

92D7:jmp &8B9B

Skip a comma at PTR #2 & get integer

This routine checks for a comma at PTR #2 and then gets the integer expression following it

Check for the comma

92DA:jsr &8AAE

Evaluate the expression

92DD:jsr &9B29

Check it is integer

92E0:jmp &92F0

Get integer operand

This routine evaluates the operand at PTR #2 and then checks it is an integer

Evaluate the operand

92E3:jsr &ADEC

Give a 'Type mismatch' if it is a string

92E6:beq &92F7

Make it into an integer if it is real

92E8:bmi &92F4

Exit

92EA:rts

Check ' = ' & evaluate integer expression

Check for equals sign and evaluate the following expression

92EB:jsr &9807

Make the flags reflect the type of the expression

92EE:lda &27

Give a 'Type mismatch' error if it is a string

92F0:beq &92F7

Exit if it is an integer

92F2:bpl &92EA

Convert to an integer if it is a real

92F4:jmp &A3E4

Direct jump to 'Type mismatch' error

92F7:jmp &8C0E

Evaluate real operand

Evaluate operand

92FA:jsr &ADEC

Give a 'Type mismatch' error if it is a string

92FD:beq &92F7

Return if it is real

92FF:bmi &92EA

Convert it to a real, and exit if it is an integer

9301:jmp &A2BE

PROC statement routine

This routine copes with calls to procedures. It loads the accumulator to indicate that a procedure is being called and joins the function calling code

Make PTR#2 point to PTR#1, which points to the character after the word PROC

9304:lda &B

9306:sta &19

9308:lda &C

930A:sta &1A

930C:lda &A

930E:sta &1B

Get the token value for 'PROC' into the accumulator, to identify where the call to the next subroutine is being made from

9310:lda #&F2

Call the code used to handle functions

9312:jsr &B197

Check for the end of the line

9315:jsr &9852

Exit

9318:jmp &8B9B

This section is part of the LOCAL routine

Zero the length of the string

931B:ldy #&3

931D:lda #&0

931F:sta (&2A),Y

Rejoin the main code

9321:beq &9341

LOCAL statement routine

The action of the LOCAL statement is quite complex, although the routine is very short

It first checks whether we are currently in a procedure or a function by examining the machine stack pointer. The machine stack pointer should be &FF if no PROC/FN is being executed, but inside a PROC/FN, it should be &FC since the return address, number of parameters and PROC/FN identifier are all stored on the stack

It then scans each variable name, creating any that do not exist, and pushes their present values onto the BASIC stack. These values will be restored at the end of the function or procedure. The variables are then cleared

The machine stack is used to hold the number of parameters, and this is incremented for each variable name found

Give a 'Not LOCAL' message if we are not inside a procedure or a function

9323:tsx

9324:cpx #&FC

9326:bcs &936B

Get next variable name

9328:jsr &9582

Exit if none found

932B:beq &9353

Save the value, address and type of variable

932D:jsr &B30D

Get the type of the variable

9330:ldy &2C

Go back if it was a string

9332:bmi &931B

Save the variable descriptor on the stack

9334:jsr &BD94

Set the IAC to zero

9337:lda #&0

9339:jsr &AED8

Set the type to integer

933C:sta &27

Assign the variable. If the variable was real, this routine will convert the contents of the IAC to a floating point number first

933E:jsr &B4B4

Get the current stack pointer

9341:tsx

Increment the number of parameters for the current procedure/function. This instruction gives rise to a very obscure bug whereby more than 255 local variables or parameters in a procedure or function cause the machine to crash

9342:inc &106,X

Update PTR #1 from PTR #2, which was used to read the variable name

9345:ldy &1B

9347:sty &A

Get the next character

9349:jsr &8A97

Go back for another name if it was a comma

934C:cmp #&2C

934E:beq &9323

Exit

9350:jmp &8B96

Exit

9353: jmp &8B98

ENDPROC statement routine

This routine checks that we are in a procedure by checking the stack depth, and the identifier byte on the stack. Assuming we are in a procedure, it checks that nothing follows the word ENDPROC, before executing an RTS instruction to pass control back to the calling routine

Check the depth of the stack

9356: tsx

9357: cpx #&FC

Give a 'No PROC' message if we are not in a procedure or function

9359: bcs &9365

Check the identifier byte

935B: lda &1FF

935E: cmp #&F2

Give a 'No PROC' message if it does not identify us to be in a procedure

9360: bne &9365

Exit via the routine to check for the end of the statement

9362: jmp &9857

'No PROC' error message

9365: brk

9366: 0D-

9367: 4E-N

9368: 6F-o

9369: 20-

936A: F2-r

'Not LOCAL' error message

936B: brk

936C: 0C-

936D: 4E-N

936E: 6F-o

936F: 74-t

9370: 20-

9371: EA-j

'Bad MODE' error message

9372: brk

9373: 19-

9374: 42-B

9375: 61-a

9376: 64-d

9377: 20-

9378: EB-k

9379: 00-

GCOL statement routine

Evaluate the modifier

937A:jsr &8821

Save it on the stack

937D:lda &2A

937F:pha

Check for a comma, and get the next integer expression

9380:jsr &92DA

Check for the end of the statement

9383:jsr &9852

Print the GCOL code

9386:lda #&12

9388:jsr &FFEE

Jump forwards to print the last two bytes

938B:jmp &93DA

COLOUR statement routine

Save the code for COLOUR

938E:lda #&11

9390:pha

Evaluate the integer expression

9391:jsr &8821

Check for the end of the statement

9394:jsr &9857

Print the colour code and the colour identifier

9397:jmp &93DA

MODE statement routine

This routine first gets the mode to be changed to, then checks if the second processor is being used. If it is being used, it just moves to the new mode and exits. If it is not being used, it checks memory requirements before changing mode

Save mode change command

939A:lda #&16

939C:pha

Evaluate the expression

939D:jsr &8821

Check for the end of the statement

93A0:jsr &9857

Get the machine higher order address. This is &FFFF for the input/output processor

93A3:jsr &BEE7

If the higher order address is not given as &FFFF, skip to the part of the routine that actually changes MODE

93A6:cpx #&FF

93A8:bne &93D7

93AA:cpy #&FF

93AC:bne &93D7

Give a 'Bad MODE' error message if there is data on the stack. This is tested for by seeing if the stack pointer is the same as HIMEM

93AE:lda &4

93B0:cmp &6

93B2:bne &9372

93B4:lda &5

93B6:cmp &7

93B8:bne &9372

Check how much memory this new mode would take

93BA:ldx &2A

93BC:lda #&85

93BE:jsr &FFF4

Give a 'Bad DIM' error message if the mode would extend below VARTOP or TOP. Both need to be tested for to allow for situations like 'LOMEM = &900:HIMEM = &D00' which is sometimes used to allow larger programs than usual by moving the variables and the stack below the program text

93C1:cpx &2

93C3:tya

93C4:sbc &3

93C6:bcc &9372

93C8:cpx &12

93CA:tya

93CB:sbc &13

93CD:bcc &9372

Save the new values for HIMEM and the stack pointer

93CF:stx &6

93D1:stx &4

93D3:sty &7

93D5:sty &5

Zero COUNT

93D7:jsr &BC28

Print the mode change command

93DA:pla

93DB:jsr &FFEE

Go to print the byte currently in the IAC

93DE:jsr &9456

Exit

93E1:jmp &8B9B

MOVE statement routine

Select PLOT 4, then jump into the DRAW routine

93E4:lda #&4

93E6:bne &93EA

DRAW statement routine

Select PLOT 5

93E8:lda #&5

Save the PLOT type

93EA:pha

Evaluate the X coordinate

93EB:jsr &9B1D

Join the PLOT routine

93EE:jmp &93FD

PLOT statement routine

Evaluate the PLOT type

93F1:jsr &8821

Save the PLOT type on the stack

93F4:lda &2A

93F6:pha

Check the presence of a comma

93F7:jsr &8AAE

Evaluate the X coordinate

93FA:jsr &9B29

Make sure it is an integer

93FD:jsr &92EE

Save it on the stack

9400:jsr &BD94

Check for a comma and evaluate the Y coordinate

9403:jsr &92DA

Check for the end of the statement

9406:jsr &9852

Print the PLOT command

9409:lda #&19

940B:jsr &FFEE

Get and print the PLOT type

940E:pla

940F:jsr &FFEE

Pull the X coordinate to locations &37,&38

9412:jsr &BE0B

Print the X coordinate

9415:lda &37

9417:jsr &FFEE

941A:lda &38

941C:jsr &FFEE

Print the LSB of the Y coordinate

941F:jsr &9456

Print the MSB of the Y coordinate

9422:lda &2B
 9424:jsr &FFEE

Exit

9427:jmp &8B9B

This routine is part of the VDU statement
 Print the MSB of the expression

942A:lda &2B
 942C:jsr &FFEE

VDU statement routine

Get the next character

942F:jsr &8A97

Exit if it is a colon

9432:cmp #&3A
 9434:beq &9453

Exit if it is a carriage return

9436:cmp #&D
 9438:beq &9453

Exit if it is the word 'ELSE'

943A:cmp #&8B
 943C:beq &9453

Move PTR#1 back one character, to allow for a statement terminator not being found

943E:dec &A

Evaluate the integer expression

9440:jsr &8821

Print it

9443:jsr &9456

Get the next character

9446:jsr &8A97

Continue the search if it is a comma

9449:cmp #&2C
 944B:beq &942F

Check for one of the other things if it is not a semi-colon

944D:cmp #&3B
 944F:bne &9432

Go back to print the MSB of the number

9451:beq &942A

Exit

9453:jmp &8B96

Print the LSB of the IAC

Get the LSB of the IAC

9456:lda &2A

Pass it to OSWRCH in a rather devious way

9458:jmp (&20E)

Search for FN/PROC in catalogue

This routine searches for the indicated procedure or function in the catalogue. It drops straight into the more general routine which follows, which finds a variable in the catalogue. On entry, (&37)+1 should point to the token for PROC or FN
 Point to the word PROC or the word FN

945B:ldy #&1

Get the word

945D:lda (&37),Y

Temporarily point to the catalogue entry for procedures

945F:ldy #&F6

If the byte indicated that we are dealing with a procedure, jump into the next routine

9461:cmp #&F2

9463:beq &946F

Else, point to the function entry

9465:ldy #&F8

Jump into the catalogue search routine

9467:bne &946F

Find name in catalogue

This routine finds the catalogue entry corresponding to the variable name starting at (&37)+1 whose length is in &39. On exit, the IAC points to the zero byte at the end of the name in the variable information block. If the zero flag is set on exit, the name is not present in the catalogue.

The routine uses two slightly different search routines, which are used alternately. This is done to reduce the amount of swapping of pointers, but it does lead to code that is quite tricky to read

Get the first character of the name

9469:ldy #&1

946B:lda (&37),Y

Multiply the character by two

946D:asl A

Use it as a pointer into the variable catalogue

946E:tay

Get the address of the first catalogue entry for a variable with the same initial letter, and save it in (&3A)

946F:lda &400,Y

9472:sta &3A

9474:lda &401,Y

9477:sta &3B

Return with Z=1 if the MSB of the link bytes of the current catalogue entry are zero, since this means that the end of the catalogue has been reached

9479:lda &3B

947B:beq &94B2

Save the link bytes of the next catalogue entry variable in (&3C)

947D:ldy #&0
947F:lda (&3A),Y
9481:sta &3C
9483:iny
9484:lda (&3A),Y
9486:sta &3D

Point to the next character of the name in the catalogue entry

9488:iny

Get the character

9489:lda (&3A),Y

Skip forwards if the zero byte at the end of the name has not been reached

948B:bne &949A

Point to the last character of the name in the variable information block

948D:dey

Y now reflects the length of the name in the catalogue entry. If this length is different from the length of the variable being sought, the search continues with the next entry

948E:cpy &39

9490:bne &94B3

Point to the zero byte after the name

9492:iny

Branch forwards. The carry flag will always be set as a result of the previous CPY instruction

9493:bcs &94A7

Get the next character from the catalogue entry

9495:iny

9496:lda (&3A),Y

If it is the zero byte marking the end of the name, move on to the next information block

9498:beq &94B3

Compare the character against the corresponding character of the name being sought

949A:cmp (&37),Y

If the characters do not match, move on to the next information block

949C:bne &94B3

If the end of the sought variable name has not been reached, continue the search

949E:cpy &39

94A0:bne &9495

Get the next character from the catalogue entry

94A2:iny

94A3:lda (&3A),Y

If it is not the zero byte marking the end of the name, move to the next information block

94A5:bne &94B3

Make the IAC point to the zero byte after the name

94A7:tya

94A8:adc &3A


```

94AA:sta &2A
94AC:lda &3B
94AE:adc #&0
94B0:sta &2B

```

Exit

```
94B2:rts
```

Control passes here when the next catalogue entry must be inspected

Exit with Z = 1 if the link bytes of the current entry indicate that there is not another information block sharing the same initial letter

```

94B3:lda &3D
94B5:beq &94B2

```

Get the new link bytes from the new catalogue entry

```

94B7:ldy #&0
94B9:lda (&3C),Y
94BB:sta &3A
94BD:iny
94BE:lda (&3C),Y
94C0:sta &3B

```

Get the next character from the catalogue entry

```

94C2:iny
94C3:lda (&3C),Y

```

Skip forwards if the character is not the zero marking the end of the name

```
94C5:bne &94D4
```

Point to the last character of the name in the information block, making Y reflect the length of the name in the catalogue entry

```
94C7:dey
```

Check it against the length of the sought variable name

```
94C8:cpy &39
```

Check the next information block if they are not the same

```
94CA:bne &9479
```

Point to the zero byte after the name

```
94CC:iny
```

Branch to exit

```
94CD:bcs &94E1
```

Get the next information block character

```

94CF:iny
94D0:lda (&3C),Y

```

Move to the next information block if the character is the end of name marker

```
94D2:beq &9479
```

Check the character against the corresponding character of the sought variable name

```
94D4:cmp (&37),Y
```

Move to the next information block if the characters do not match

```
94D6:bne &9479
```

Continue the search if the end of the sought variable name has not been reached

```
94D8:cpy &39
```

94DA:bne &94CF

Move on to the next information block if the character in the information block is not the zero byte marking the end of the name

94DC:iny

94DD:lda (&3C),Y

94DF:bne &9479

Make the IAC point to the zero byte at the end of the information block

94E1:tya

94E2:adc &3C

94E4:sta &2A

94E6:lda &3D

94E8:adc #&0

94EA:sta &2B

94EC:rts

Create a catalogue entry for a PROC/FN

The entry parameters are the same as the previous routine.

Get the first letter of the name of the PROC/FN, which will either be the token for FN or the token for PROC

94ED:ldy #&1

94EF:lda (&37),Y

Put the token into X

94F1:tax

Load the catalogue offset for procedures

94F2:lda #&F6

Enter the 'create variable' routine if the token indicated a procedure

94F4:cpx #&F2

94F6:beq &9501

Otherwise, load the catalogue offset for functions and jump into the 'create variable' routine

94F8:lda #&F8

94FA:bne &9501

Create variable catalogue entry

The parameters for this routine are the same as for the 'find name in catalogue' routine. On exit, VARTOP points to the start of the newly created variable information block and Y is the offset from VARTOP of the last letter of the name of the variable

Get the first letter of the variable

94FC:ldy #&1

94FE:lda (&37),Y

Multiply the letter by two

9500:asl A

Use it to construct a pointer into the variable catalogue

9501:sta &3A

9503:lda #&4

9505:sta &3B

Get the MSB of the link bytes from the current information block

9507:lda (&3A),Y

Go forwards to create the variable if the byte is zero, since this indicates the end of the list of variables sharing the same initial letter as the one being created

9509:beq &9516

Save the MSB of the link bytes in X

950B:tax

Point to the LSB of the link bytes

950C:dey

Get the LSB of the link bytes

950D:lda (&3A),Y

Point to the information block indicated by the link bytes

950F:sta &3A

9511:stx &3B

Make Y point to the MSB of the link bytes of the new information block

9513:iny

Loop back

9514:bpl &9507

When the last information block has been found, use VARTOP as the new link bytes

9516:lda &3

9518:sta (&3A),Y

951A:lda &2

951C:dey

951D:sta (&3A),Y

Get zero into the accumulator

951F:tya

Get 1 into Y, to point to the MSB of the new link bytes of the information block under creation

9520:iny

Save zero as the MSB, to indicate this is the last information block with the current initial letter

9521:sta (&2),Y

Exit if the name has only one letter. Otherwise, the rest of the name has to be copied into the block

9523:cpy &39

9525:beq &9558

Point to the next character of the name

9527:iny

Get the current character

9528:lda (&37),Y

Store it in the information block

952A:sta (&2),Y

If the end of the name has not been reached, continue this process

952C:cpy &39

952E:bne &9527

9530:rts

Clear a newly created variable

This routine zeros the byte after the name in an information block, together with the value bytes associated with the type of variable in the information block. The number of bytes that need to be cleared is passed in the X register, while the Y register offset from VARTOP must point to the last character of the name of the variable. After the bytes have been cleared, VARTOP is made to point to the first byte that was not cleared

Load the accumulator with zero

9531:lda #&0

Point to the next byte in the information block

9533:iny

Zero it

9534:sta (&2),Y

Decrement the counter of the number of bytes to be zeroed

9536:dex

Continue the process until there are no more bytes to zero

9537:bne &9533

Add Y + 1 to VARTOP

9539:sec

953A:tya

953B:adc &2

953D:bcc &9541

953F:inc &3

Compare the MSB of this new VARTOP to the MSB of the stack pointer

9541:ldy &3

9543:cpy &5

If VARTOP is lower, exit

9545:bcc &9556

Do not bother to check the LSB if the MSBs are not the same

9547:bne &954D

Check the LSB of VARTOP against the LSB of the stack pointer

9549:cmp &4

Exit if VARTOP is lower

954B:bcc &9556

Zero the MSB of the link byte of this partially created information block, to stop the block existing

954D:lda #&0

954F:ldy #&1

9551:sta (&3A),Y

Give a 'No room' error

9553: jmp &8CB7

Store the new LSB of VARTOP

9556: sta &2

Exit

9558: rts

Get an array name

This routine searches for an array name starting at (&37)+1. On exit, Y and (&37) point to the first character that could not be interpreted as part of the array name. In addition, A contains this character. X is incremented for each character that is scanned

Point to the first character of the name

9559: ldy #&1

Get the current character

955B: lda (&37), Y

Exit if the character is less than '0'

955D: cmp #&30

955F: bcc &9579

Jump forwards if the character is greater than or equal to '@'

9561: cmp #&40

9563: bcs &9571

Exit if the character is greater than '9'

9565: cmp #&3A

9567: bcs &9579

If control gets here, the character is a number. Of course, numbers are only allowed after the first character of the array name, so a check is made to ensure we are past the first character

9569: cpy #&1

956B: beq &9579

Point to the next character

956D: inc

956E: iny

Get the next character

956F: bne &955B

Jump forwards if the character is greater than or equal to '_'

9571: cmp #&5F

9573: bcs &957A

Continue the search if the character is less than '['

9575: cmp #&5B

9577: bcc &956D

Exit

9579: rts

Continue if the character is less than '{'

957A: cmp #&7B

957C:bcc &956D

Exit

957E:rts

Clear the variable

957F:jsr &9531

Get a variable, creating it if needed

This subroutine skips spaces then gets a variable name. If the variable does not exist, it is created

Search for the variable name

9582:jsr &95C9

Exit if the variable exists

9585:bne &95A4

Exit if the variable is invalid

9587:bcs &95A4

Create the variable

9589:jsr &94FC

Indicate five bytes are to be cleared. This is satisfactory for integers and strings

958C:ldx #&5

Check the type of the variable

958E:cpx &2C

Go back to clear the variable if it is not real

9590:bne &957F

If it is real, indicate six bytes are to be cleared and go back to clear them

9592:inx

9593:bne &957F

Check for '!

9595:cmp #&21

9597:beq &95A5

Check for '\$'

9599:cmp #&24

959B:beq &95B0

Check for '?'

959D:eor #&3F

959F:beq &95A7

Set the zero flag to indicate that the variable was not found

95A1:lda #&0

Set the carry flag to indicate that the variable name is invalid

95A3:sec

Exit

95A4:rts

Unary '!

Set the type to 4

95A5:lda #&4

Unary '?'

When this routine is entered, the accumulator will always contain zero

Save the number of bytes

95A7:pha

Increment pointer past '!' or '?'

95A8:inc &1B

Get the integer expression following the operator

95AA:jsr &92E3

Join the code that deals with the binary operators

95AD:jmp &969F

\$< address>

This routine copes with variables of the form '\$F%'

Increment the pointer past the dollar sign

95B0:inc &1B

Evaluate the following integer expression

95B2:jsr &92E3

Give a '\$ range' error if the address of the string is less than 256

95B5:lda &2B

95B7:beq &95BF

Set the type to &80; also set Z=0 to indicate a valid variable

95B9:lda #&80

95BB:sta &2C

Indicate a string variable

95BD:sec

Exit

95BE:rts

95BF:brk

95C0:08-

95C1:24-\$

95C2:20-

95C3:72-r

95C4:61-a

95C5:6E-n

95C6:67-g

95C7:65-e

95C8:00-

Search for variable name at PTR # 1

This routine loads PTR # 2 with PTR # 1 and skips spaces before searching for a variable name

Copy PTR # 1 to PTR # 2

```

95C9:lda &B
95CB:sta &19
95CD:lda &C
95CF:sta &1A
95D1:ldy &A

```

Move one character back to allow for the first INY instruction

```
95D3:dey
```

Point to the next instruction

```
95D4:iny
```

Save the offset

```
95D5:sty &1B
```

Get the character

```
95D7:lda (&19),Y
```

Increment the offset if it is a space

```
95D9:cmp #&20
95DB:beq &95D4
```

Search for variable name at PTR #2

This routine should be entered with the accumulator containing the first character of the alleged variable. The routine is optimized towards identifying the resident integer variables as fast as it can. On exit, the address of the value of the variable is stored in &2A,&2B, while its type is stored in &2C, using the same conventions as the CALL statement. If the zero flag is set on exit, the variable does not exist. The carry flag will be set if the variable was invalid, but reset if the variable was valid but undefined. If the zero flag is not set, a valid, defined variable was found. The carry flag will be set if the variable is a string variable

This can be summarized as follows:

Z=0;C=0; A defined numeric variable was found

Z=0;C=1; A defined string variable was found

Z=1;C=0; A valid but undefined variable was found

Z=1;C=1; An invalid variable was found

Confusingly, the '!', '\$' and '?' operators can appear in variable names; thus, 'A%!gh%' is a variable name, not an expression. If this was not true, one would not be able to write things like 'CALL address,param,array?element'

Skip backwards to &9595 if the character is less than '@'. If it is less than '@', it can only be a leading '\$', '?' or '!'.

```
95DD:cmp #&40
95DF:bcc &9595
```

Go forwards to &95FF if the character is greater than or equal to '[' - this leaves us with an uppercase letter

```
95E1:cmp #&5B
95E3:bcs &95FF
```

Multiply the letter by four. This is to make it an index into the page four integer variable storage area

95E5:asl A
95E6:asl A
 Save the address of the variable
95E7:sta &2A
95E9:lda #&4
95EB:sta &2B
 Get the next character
95ED:iny
95EE:lda (&19),Y
 Point to the next character
95F0:iny
 If the next character is not a '%', skip forwards
95F1:cmp #&25
95F3:bne &95FF
 Set the type of the variable to integer
95F5:ldx #&4
95F7:stx &2C
 Get the next character
95F9:lda (&19),Y
 If it is not a '(', skip to &9665
95FB:cmp #&28
95FD:bne &9665
 This is the main variable scanning routine. It can cope with everything except the resident integer variables.
 Set the type to real
95FF:ldx #&5
9601:stx &2C
 Add the offset of PTR #2 to PTR #2, leaving the result in A(lsb) and X(msb)
9603:lda &1B
9605:clc
9606:adc &19
9608:ldx &1A
960A:bcc &960E
960C:inx
960D:clc
 Move to (&37), subtracting one in the process. This makes (&37) a pointer to the character before the first character of the variable name
960E:sbx #&0
9610:sta &37
9612:bcs &9615
9614:dex
9615:stx &38
 Get the offset of the start of the variable. This will be incremented as each letter is scanned, to keep track of the location of the final letter of the name
9617:ldx &1B

Let Y be the offset to the pointer at (&37)

9619:ldy #&1

Get the next character

961B:lda (&37),Y

Goto &962D if the character is greater than or equal to 'A'

961D:cmp #&41

961F:bcs &962D

Goto &9641 if the character is less than '0'

9621:cmp #&30

9623:bcc &9641

Goto &9641 if the character is greater than '0'

9625:cmp #&3A

9627:bcs &9641

Point to the next character

9629:inx

962A:iny

Go back and get the next character

962B:bne &961B

Goto &9635 if the character is greater than 'Z'

962D:cmp #&5B

962F:bcs &9635

Deal with the next character

9631:inx

9632:iny

9633:bne &961B

Goto &9641 if the character is less than '_'

9635:cmp #&5F

9637:bcc &9641

Goto &9641 if the character is greater than 'z'

9639:cmp #&7B

963B:bcs &9641

Deal with the next character

963D:inx

963E:iny

963F:bne &961B

When control passes to here, as many characters as possible have been scanned

If Y contains 1, the character that could not be used in the name was the first character, so the variable was invalid. In this case, control passes to &9673, where the zero flag and carry flags are set.

9641:dey

9642:beq &9673

Goto &96AF if the character that could not be scanned was '\$'

9644:cmp #&24

9646:beq &96AF

Skip forwards if it was not a '%'

9648:cmp #&25

964A:bne &9654

Decrement the type to integer

964C:dec &2C

Allow for the spurious DEY instruction at &9641

964E:iny

Get the next character

964F:inx

9650:iny

9651:lda (&37),Y

Decrement Y again, since it must be maintained at one character to the left of its real position

9653:dey

Save the length of the name

9654:sty &39

Branch forwards if an array is being dealt with

9656:cmp #&28

9658:beq &96A6

Find the address of the variable

965A:jsr &9469

Exit with Z = 1, C = 0 if the variable is undefined

965D:beq &9677

Update the offset of PTR #2

965F:stx &1B

Get the current character at PTR #2

9661:ldy &1B

9663:lda (&19),Y

Goto &967F if the character that could not be scanned was '!

9665:cmp #&21

9667:beq &967F

Goto &967B if the character that could not be scanned was '?'

9669:cmp #&3F

966B:beq &967B

Indicate that the variable is numeric

966D:clc

Update the offset of PTR #2

966E:sty &1B

Set Z = 0 to indicate that a valid, defined variable was found

9670:lda #&FF

Exit

9672:rts

Set Z = 1 and C = 1 to indicate that an invalid variable was found

9673:lda #&0

9675:sec

9676:rts

Set Z = 1 and C = 0 to indicate that the variable was valid but is not defined

9677:lda #&0

9679:clc

967A:rts

Binary '?'

Set the type to zero

967B:lda #&0

Join the code to deal with '!

967D:beq &9681

Binary '!'

Set the type to four

967F:lda #&4

Save the type

9681:pha

Increment PTR # 2 past the '?' or '!'

9682:iny

9683:sty &1B

Get the value of the variable scanned so far

9685:jsr &B32C

Make it into an integer

9688:jsr &92F0

Save the address indicate by the variable

968B:lda &2B

968D:pha

968E:lda &2A

9690:pha

Get the integer operand following the '!' or '?'

9691:jsr &92E3

Add the address on the stack to the value of the operand

9694:clc

9695:pla

9696:adc &2A

9698:sta &2A

969A:pla

969B:adc &2B

969D:sta &2B

Retrieve the type

969F:pla

Save the type

96A0:sta &2C

Indicate a valid numeric variable was found

96A2:clc

96A3:lda #&FF

Exit

96A5:rts

Increment past the '('

96A6:inx**96A7:inc &39**

Deal with the array

96A9:jsr &96DF

Check for the indirection operators

96AC:jmp &9661

Increment past the '\$' at the end of the name

96AF:inx**96B0:iny**

Save the length of the name

96B1:sty &39

Point to the next character

96B3:iny

Decrement the type to 4

96B4:dec &2C

Get the next character

96B6:lda (&37),Y

Skip to &96C9 if it is a '('

96B8:cmp #&28**96BA:beq &96C9**

Find the address of the variable

96BC:jsr &9469

Exit if the variable is undefined

96BF:beq &9677

Update the offset of PTR#2

96C1:stx &1B

Indicate a type of &81; set Z=0 and C=1 to indicate a valid string variable has been found

96C3:lda #&81**96C5:sta &2C****96C7:sec**

Exit

96C8:rts

Increment past the '('

96C9:inx**96CA:sty &39**

Decrement the type to 3

96CC:dec &2C

Deal with the array

96CE:jsr &96DF

Set the type to &81; also set Z=0 and C=1 to indicate that a valid string variable was found

96D1:lda #&81**96D3:sta &2C****96D5:sec**

Exit

96D6:rts**96D7:brk****96D8:0E-****96D9:41-A****96DA:72-r****96DB:72-r****96DC:61-a****96DD:79-y****96DE:00-****Deal with array**

This routine copes with an array appearing in place of a normal variable in an expression or elsewhere. It is only called from the main 'find variable' routine

Find the address of the variable. This will be the address of the byte which gives the number of elements in the array

96DF:jsr &9469

Give an 'Array' error message if the variable does not exist

96E2:beq &96D7

Update the offset of PTR#2

96E4:stx &1B

Save the type and address of the base of the array on the machine stack

96E6:lda &2C**96E8:pha****96E9:lda &2A****96EB:pha****96EC:lda &2B****96EE:pha**

Get the number of elements of the array (this is stored as $2*n+1$, where 'n' is the number of dimensions)

96EF:ldy #&0**96F1:lda (&2A),Y**

If the array is one dimensional, skip to &976C

96F3:cmp #&4**96F5:bcc &976C**

Zero the IAC

96F7:tya

96F8:jsr &AED8

Save the pointer to the current subscript index

96FB:lda #&1

96FD:sta &2D

Save the IAC and the pointer on the BASIC stack

96FF:jsr &BD94

Evaluate the next array index

9702:jsr &92DD

Check the presence of a comma, giving an 'Array' error message if one is not present

9705:inc &1B

9707:cpx #&2C

9709:bne &96D7

Pull the IAC to &39 onwards

970B:ldx #&39

970D:jsr &BE0D

Get the offset of the current index

9710:ldy &3C

Retrieve the base address of the array

9712:pla

9713:sta &38

9715:pla

9716:sta &37

Place it back on the stack for safe keeping

9718:pha

9719:lda &38

971B:pha

Check the current index against its limit

971C:jsr &97BA

Save the pointer to the current element

971F:sty &2D

Save the next subscript limit

9721:lda (&37),Y

9723:sta &3F

9725:iny

9726:lda (&37),Y

9728:sta &40

Add present subscript to 'total subscript' (the 'total subscript' is the subscript into the array when it is treated as a single dimension array)

972A:lda &2A

972C:adc &39

972E:sta &2A

9730:lda &2B

9732:adc &3A

9734:sta &2B

Multiply the total subscript by the next subscript limit

9736:jsr &9236
 Get the offset of the last subscript
9739:ldy #&0
973B:sec
973C:lda (&37),Y
 Subtract the offset of the current subscript
973E:sub &2D
 If there is more than one subscript left, continue the above process
9740:cmp #&3
9742:bcs &96FF
 Save the IAC on the machine stack
9744:jsr &BD94
 Evaluate expression and check for a right hand bracket
9747:jsr &AE56
 Check expression is integer
974A:jsr &92F0
 Get base address of array
974D:pla
974E:sta &38
9750:pla
9751:sta &37
 Pull IAC to &39 onwards
9753:ldx #&39
9755:jsr &BE0D
 Get the offset of the current subscript limit
9758:ldy &3C
 Check the subscript against its limit
975A:jsr &97BA
 Add new subscript to 'total subscript'
975D:clc
975E:lda &39
9760:adc &2A
9762:sta &2A
9764:lda &3A
9766:adc &2B
9768:sta &2B
 Join the following code
976A:bcc &977D

Deal with single dimension arrays
 Get expression and check for a ')'
 976C:jsr &AE56
 Ensure the expression gave an integer result
 976F:jsr &92F0
 Retrieve the base address of the array

9772:pla

9773:sta &38

9775:pla

9776:sta &37

Check the subscript

9778:ldy #&1

977A:jsr &97BA

Get the type of the array

977D:pla

Save it in &2C

977E:sta &2C

Jump to &979B if we are not dealing with a real array

9780:cmp #&5

9782:bne &979B

This section of code multiplies the 'total subscript' by five, to gain the number of bytes into the array of the selected element

Get the 'total subscript'

9784:ldx &2B

9786:lda &2A

Multiply it by two

9788:asl &2A

978A:rol &2B

Multiply by two again (*4 altogether)

978C:asl &2A

978E:rol &2B

Add original contents of total subscript (*5 altogether)

9790:adc &2A

9792:sta &2A

9794:txa

9795:adc &2B

9797:sta &2B

Skip the code for integers and strings

9799:bcc &97A3

Multiply the total subscript by four for integer and string arrays

979B:asl &2A

979D:rol &2B

979F:asl &2A

97A1:rol &2B

Add the length of the preamble to the array

97A3:tya

97A4:adc &2A

97A6:sta &2A

97A8:bcc &97AD

97AA:inc &2B

97AC:clc

Add the base address of the array to get the actual address of the element

```

97AD:lda &37
97AF:adc &2A
97B1:sta &2A
97B3:lda &38
97B5:adc &2B
97B7:sta &2B

```

Exit

```

97B9:rts

```

Check array subscript

This routine checks the array subscript in the IAC against the limit pointed to by (&37),Y

Ensure the subscript is less than 16384

```

97BA:lda &2B
97BC:and #&C0
97BE:ora &2C
97C0:ora &2D

```

Give a 'Subscript' error if it is not

```

97C2:bne &97D1

```

Check the subscript against the limit, again giving a 'Subscript' error if the subscript is too big

```

97C4:lda &2A
97C6:cmp (&37),Y
97C8:iny
97C9:lda &2B
97CB:sbc (&37),Y
97CD:bcs &97D1
97CF:iny
97D0:rts

```

```

97D1:brk
97D2:0F-
97D3:53-S
97D4:75-u
97D5:62-b
97D6:73-s
97D7:63-c
97D8:72-r
97D9:69-i
97DA:70-p
97DB:74-t
97DC:00-

```


Check for line number

This routine checks for the presence of a line number at PTR#1. On exit, C=0 indicates that no line number was found, C=1 indicates that the line number is in the IAC

Increment the offset of PTR#1

97DD:inc &A

Get the current character from PTR#1

97DF:ldy &A

97E1:lda (&B),Y

Get a new character if it was a space

97E3:cmp #&20

97E5:beq &97DD

Exit if the next character is not a line number token

97E7:cmp #&8D

97E9:bne &9805

Decode line number

This routine decodes the three line number bytes at PTR#1. The routine is practically impossible to understand without referring to the format of tokenised line numbers

Point to the first line number byte

97EB:iny

Get the first byte

97EC:lda (&B),Y

Multiply by two to get the 128 bit and the 64 bit into bits 7 and 6 respectively

97EE:asl A

97EF:asl A

Save the 16384 bit in X

97F0:tax

Mask out all the bits except the 128 and 64 bits

97F1:and #&C0

Point to the next byte

97F3:iny

Use the EOR action to include the 128 and 64 bits

97F4:eor (&B),Y

Save this as the LSB

97F6:sta &2A

Move the 16384 bit into bit 6

97F8:txa

97F9:asl A

97FA:asl A

97FB:iny

Include the final byte of the tokenised number

97FC:eor (&B),Y

Save it as the MSB of the line number

97FE:sta &2B

Update the offset of PTR # 1

9800:iny

9801:sty &A

Indicate a line number was found and exit

9803:sec

9804:rts

Indicate a line number was not found and exit

9805:clc

9806:rts

Set PTR # 2 = PTR # 1 and enter next routine

9807:lda &B

9809:sta &19

980B:lda &C

980D:sta &1A

980F:lda &A

9811:sta &1B

Check for ' = '

Evaluate expression

Check for end of statement

Skip spaces

9813:ldy &1B

9815:inc &1B

9817:lda (&19),Y

9819:cmp #&20

981B:beq &9813

Join routine to evaluate expression and check for the end of the statement if the next character is '='. If the character is something else, fall into 'Mistake' error message

981D:cmp #&3D

981F:beq &9849

9821:brk

9822:04-

9823:4D-M

9824:69-i

9825:73-s

9826:74-t

9827:61-a

9828:6B-k

9829:65-e

982A:brk

982B:10-

982C:53-S

982D:79-y
982E:6E-n
982F:74-t
9830:61-a
9831:78-x
9832:20-
9833:65-e
9834:72-r
9835:72-r
9836:6F-o
9837:72-r
9838:brk
9839:11-
983A:45-E
983B:73-s
983C:63-c
983D:61-a
983E:70-p
983F:65-e
9840:00-

Check for '=' at PTR#2

Get the next character

9841:jsr &8A8C

Give a 'Mistake' error if it is not an '='

9844:cmp #&3D

9846:bne &9821

Exit

9848:rts

Evaluate expression

Check for the end of the line

Evaluate expression

9849:jsr &9B29

Place the next character in A

984C:txa

Get the offset of PTR#2

984D:ldy &1B

Check for the end of the statement

984F:jmp &9861

9852:ldy &1B

9854:jmp &9859

Check for the end of the statement

This routine checks for the end of the current statement by ensuring that the next character is a colon, a carriage return or ELSE. Then PTR#1 is updated to point directly to the terminating character. Finally, the escape flag is checked

Skip spaces

```

9857:ldy &A
9859:dey
985A:iny
985B:lda (&B),Y
985D:cmp #&20
985F:beq &985A

```

Jump forwards if the next character is a colon

```

9861:cmp #&3A
9863:beq &986D

```

Jump forwards if the next character is a carriage return

```

9865:cmp #&D
9867:beq &986D

```

Give a 'Syntax error' if the next character is not 'ELSE'

```

9869:cmp #&8B
986B:bne &982A

```

Add the offset of PTR # 1 to PTR # 1 and point to the character after ELSE, the colon or the carriage return

```

986D:clc
986E:tya
986F:adc &B
9871:sta &B
9873:bcc &9877
9875:inc &C
9877:ldy #&1
9879:sty &A

```

Give an 'Escape' error if the escape flag is set

```

987B:bit &FF
987D:bmi &9838

```

Exit

```

987F:rts

```

Move to next statement

This routine makes PTR # 1 point to the start of the next statement, by moving to the next line if necessary. If immediate mode is being used, the system executes a cold start rather than moving on to the next line. If a new line has to be executed, the TRACE output is given if it is selected

Check for the end of the statement

```

9880:jsr &9857

```

Get the character that caused the end of the statement

```

9883:dey
9884:lda (&B),Y

```

Exit if it was a colon

```

9886:cmp #&3A
9888:beq &987F

```

Do a cold start if immediate mode is in effect

988A:lda &C

988C:cmp #&7

988E:beq &98BC

Get the MSB of the line number of the next line

9890:iny

9891:lda (&B),Y

Execute a cold start if the end of the program is reached - indicated by the line number being over 32767

9893:bmi &98BC

Skip doing the TRACE output if TRACE is turned off

9895:lda &20

9897:beq &98AC

Save the current text offset on the stack

9899:tya

989A:pha

Get the LSB of the line number of the line to be TRACEd and put it on the stack

989B:iny

989C:lda (&B),Y

989E:pha

Put the LSB into the Y register

989F:dey

98A0:lda (&B),Y

98A2:tay

Get the MSB back into A

98A3:pla

Place the line number in the IAC

98A4:jsr &AEEA

Do the TRACE output if required

98A7:jsr &9905

Get the offset back

98AA:pla

98AB:tay

Point to the LSB of the line number

98AC:iny

Make PTR #1 point to the start of the line by adding one more than the offset of the LSB of the line number

98AD:sec

98AE:tya

98AF:adc &B

98B1:sta &B

98B3:bcc &98B7

98B5:inc &C

Set the offset to 1

98B7:ldy #&1

98B9:sty &A

Exit

98BB:rts

Direct jump to cold start

98BC:jmp &8AF6

Direct jump to the 'Type mismatch' error

98BF:jmp &8C0E

IF statement routine

The IF statement first evaluates the condition in the form of an integer expression. If the result is zero, the remainder of the current line is scanned for the word 'ELSE'. If it is found, the statements following it are executed. If no 'ELSE' is found, control simply passes to the next line. If the result of the expression is non-zero, the statements after the word 'THEN' are executed. The complicating factors are the possible omission of the word 'THEN' and the possible presence of a line number after either 'THEN' or 'ELSE', as in 'IF A=2 THEN 320'

Evaluate the expression

98C2:jsr &9B1D

Give a 'Type mismatch' error if the result of the expression is a string

98C5:beq &98BF

Skip converting the expression to an integer if it already is an integer

98C7:bpl &98CC

Convert the contents of FAC#1 to an integer in the IAC

98C9:jsr &A3E4

Update the offset of PTR#1 with that of PTR#2

98CC:ldy &1B

98CE:sty &A

Check the contents of the IAC to see if it is zero

98D0:lda &2A

98D2:ora &2B

98D4:ora &2C

98D6:ora &2D

Goto &98F1 if the IAC does contain zero

98D8:beq &98F1

Skip the next instruction if 'THEN' is present

98DA:cpx #&8C

98DC:beq &98E1

Continue executing from the end of the expression. This is how statements like 'IF A=2 GOTO 320' are dealt with

98DE:jmp &8BA3

Increment past the word 'THEN'

98E1:inc &A

Check for a line number token

98E3:jsr &97DF

Simply execute the statement following 'THEN' if the line number was not present

98E6:bcc &98DE

Check that the line exists

98E8:jsr &B9AF

Make PTR #1 point directly to the end of the line number

98EB:jsr &9877

Join the code for 'GOTO'

98EE:jmp &B8D2

Get the next character

98F1:ldy &A

98F3:lda (&B),Y

Exit if it is a carriage return

98F5:cmp #&D

98F7:beq &9902

Point the next character

98F9:iny

98FA:cmp #&8B

Continue the search if 'ELSE' not present

98FC:bne &98F3

Update the offset of PTR #1

98FE:sty &A

Join the code for 'THEN'

9900:beq &98E3

Exit

9902:jmp &8B87

Do TRACE output if required

Check whether the current line number is greater than the TRACE limit

9905:lda &2A

9907:cmp &21

9909:lda &2B

990B:sbc &22

Return if it is

990D:bcs &98BB

Output a '['

990F:lda #&5B

9911:jsr &B558

Print the line number

9914:jsr &991F

Output a ']'

9917:lda #&5D

9919:jsr &B558

Exit and print a space

991C:jmp &B565

Print the IAC as a 16 bit number

This routine has two entry points: if entered at &991F the IAC is printed with no leading spaces, but if entered at &9923, a field width of five is used. The latter setting is used for printing line numbers

The method used is to find out how many 10000s, 1000s, 100s, 10s and 1s can be subtracted from the number. The five digits of the number will be given by these numbers. The powers of ten are stored in two tables. The LSB of the table can be found at &996B, while the MSB table starts at &99B9

Set the field width to zero

991F:lda #&0

Skip setting the field width to five

9921:beq &9925

Select a field width of five

9923:lda #&5

Save the field width

9925:sta &14

Start by pointing to the leftmost digit (the digits will be built up from &3F onwards)

9927:ldx #&4

Zero the current digit

9929:lda #&0

992B:sta &3F,X

Subtract the current LSB table value from the IAC

992D:sec

992E:lda &2A

9930:sbc &996B,X

Saving the result in Y

9933:tay

Subtract the MSB table value as well

9934:lda &2B

9936:sbc &99B9,X

If underflow is detected, exit the loop

9939:bcc &9943

Save the new value of the IAC

993B:sta &2B

993D:sty &2A

Increment the current digit

993F:inc &3F,X

Go back to try another trial subtraction

9941:bne &992E

Point to the next digit

9943:dex

Continue the process until the last digit has been dealt with

9944:bpl &9929

Point to one position after the leftmost digit

9946:ldx #&5

Point to the next lower digit

9948:dex

Exit if we have reached the end of the number

9949:beq &994F

Get the current digit

994B:lda &3F,X

Continue if it is zero

994D:beq &9948

X now contains the index of the first non-zero digit, or the last digit if the entire IAC was zero. This allows us to print the number without leading zeros

994F:stx &37

Skip printing leading spaces if the field width is set to zero

9951:lda &14**9953:beq &9960**

Take the first digit position away from the total length assigned to the number

9955:sub &37

Skip printing any leading spaces if the result is zero

9957:beq &9960

Print the required number of spaces

9959:tay**995A:jsr &B565****995D:dey****995E:bne &995A**

Get the first digit

9960:lda &3F,X

Include the ASCII offset

9962:ora #&30

Print it

9964:jsr &B558

Print the next digit if there are any left

9967:dex**9968:bpl &9960**

Exit

996A:rts

This is the table of the LSBs of the powers of ten

996B:01-**996C:0A-****996D:64-d****996E:E8-h****996F:10-****Search for line in program**

This routine is entered with a line number in the IAC. On exit, if the line does not exist,

the carry flag will be set. If it does exist, (&3D) contain one less than the address of the text of the line

Set (&3D) to PAGE

9970:ldy #&0

9972:sty &3D

9974:lda &18

9976:sta &3E

Get the MSB of the current line number

9978:ldy #&1

997A:lda (&3D),Y

Check it against the MSB of the line number being sought

997C:cmp &2B

Goto &998E if the MSB of the current line number is greater than or equal to the MSB of the line number being sought

997E:bcs &998E

Get the length of the current line

9980:ldy #&3

9982:lda (&3D),Y

Add it to the pointer

9984:adc &3D

9986:sta &3D

9988:bcc &9978

998A:inc &3E

Continue the search

998C:bcs &9978

If the MSBs are not actually equal, exit with C = 1

998E:bne &99A4

Get the LSB of the current line

9990:ldy #&2

9992:lda (&3D),Y

Check it against the LSB of the line being sought

9994:cmp &2A

Continue the search if the current line is too small

9996:bcc &9980

Exit with C = 1 if the numbers are not actually equal

9998:bne &99A4

Add the 3 to the pointer

999A:tya

999B:adc &3D

999D:sta &3D

999F:bcc &99A4

99A1:inc &3E

99A3:clc

Set Y to 2 as the offset

99A4:ldy #&2
Exit

99A6:rts
99A7:brk
99A8:12-
99A9:44-D
99AA:69-i
99AB:76-v
99AC:69-i
99AD:73-s
99AE:69-i
99AF:6F-o
99B0:6E-n
99B1:20-
99B2:62-b
99B3:79-y
99B4:20-
99B5:7A-z
99B6:65-e
99B7:72-r
99B8:6F-o

This is the table of the MSBs of the powers of ten

99B9:00-
99BA:00-
99BB:00-
99BC:03-
99BD:27-'

MOD/DIV operator routine

This routine carries out integer division. It is used by both MOD and DIV. On entry, the first parameter should be in the IAC or FAC#1

The method used is similar to that we discussed earlier, with a number of refinements. Most noticeably, the dividend is shifted on its own until a set bit appears at the leftmost position

Ensure the dividend is an integer

99BE:tay
99BF:jsr &92F0

Save the sign of the dividend

99C2:lda &2D
99C4:pha

Take the absolute value of the dividend

99C5:jsr &AD71

Push the dividend and call LEVEL 6 to get the other operand

99C8:jsr &9E1D

Save the next character

99CB:stx &27

Ensure the divisor is an integer

99CD:tay

99CE:jsr &92F0

Get the sign of the dividend back

99D1:pla

Save it in &38

99D2:sta &38

Exclusive-OR it with the sign of the divisor

99D4:eor &2D

Save the result in &37

99D6:sta &37

Take the absolute value of the divisor

99D8:jsr &AD71

Copy the dividend to &39 onwards

99DB:ldx #&39

99DD:jsr &BE0D

Zero the accumulator

99E0:sty &3D

99E2:sty &3E

99E4:sty &3F

99E6:sty &40

Give a 'Division by zero' error if the divisor is zero

99E8:lda &2D

99EA:ora &2A

99EC:ora &2B

99EE:ora &2C

99F0:beq &99A7

Set up 31 iterations through the main loop

99F2:ldy #&20

Exit if enough iterations have taken place

99F4:dey

99F5:beq &9A38

Shift the dividend left

99F7:asl &39

99F9:rol &3A

99FB:rol &3B

99FD:rol &3C

If the most significant bit is still zero, repeat the process

99FF:bpl &99F4

Shift the dividend left

9A01:rol &39

9A03:rol &3A

9A05:rol &3B

9A07:rol &3C

Shift the carry out of the dividend into the accumulator

9A09:rol &3D

9A0B:rol &3E

9A0D:rol &3F

9A0F:rol &40

Subtract the LSB of the divisor from the LSB of the accumulator, pushing the result on the stack

9A11:sec

9A12:lda &3D

9A14:cbc &2A

9A16:pha

Subtract the next bytes, pushing the result on the stack

9A17:lda &3E

9A19:cbc &2B

9A1B:pha

Subtract the next bytes, leaving the result in the X register

9A1C:lda &3F

9A1E:cbc &2C

9A20:tax

Subtract the MSBs

9A21:lda &40

9A23:cbc &2D

If the subtraction did not 'go', discard the result

9A25:bcc &9A33

Save the result of the subtraction in the accumulator

9A27:sta &40

9A29:stx &3F

9A2B:pla

9A2C:sta &3E

9A2E:pla

9A2F:sta &3D

Skip clearing the stack

9A31:bcs &9A35

Discard the two bytes of the result on the stack

9A33:pla

9A34:pla

Continue the loop

9A35:dey

9A36:bne &9A01

Exit

9A38:rts

This routine is called when a comparison is attempted where the first operand is an integer and the second is real. On entry, the integer is on the stack and the real number is in FAC#1

Save the next character

9A39:stx &27

Pull the integer from the stack

9A3B:jsr &BDEA

Save the floating point number

9A3E:jsr &BD51

Convert the integer to a real number

9A41:jsr &A2BE

Copy it to FAC #2

9A44:jsr &A21E

Discard the real number on the stack, leaving (&4B) pointing to it

9A47:jsr &BD7E

Unpack FAC #1 from (&4B)

9A4A:jsr &A3B5

Join the main floating point comparison routine

9A4D:jmp &9A62

Floating point comparison

Save the first operand on the stack

9A50:jsr &BD51

Call LEVEL 4

9A53:jsr &9C42

Save the next character

9A56:stx &27

Make sure the second operand is a real number

9A58:tay

9A59:jsr &92FD

Discard the floating point number on the stack, leaving (&4B) pointing to it

9A5C:jsr &BD7E

Unpack the number at (&4B) to FAC #2

9A5F:jsr &A34E

Get the next character back

9A62:ldx &27

Set up Y

9A64:ldy #&0

Flip the sign of FAC #2

9A66:lda &3B

9A68:and #&80

9A6A:sta &3B

Flip the sign of FAC #1

9A6C:lda &2E

9A6E:and #&80

Compare the two signs

9A70:cmp &3B

Exit if they are different. The signs had to be inverted to ensure that if an exit were made at this point, the flags would show the correct values

9A72:bne &9A92

Compare the exponents

9A74:lda &3D

9A76:cmp &30

9A78:bne &9A93

Compare the mantissas

9A7A:lda &3E

9A7C:cmp &31

9A7E:bne &9A93

9A80:lda &3F

9A82:cmp &32

9A84:bne &9A93

9A86:lda &40

9A88:cmp &33

9A8A:bne &9A93

9A8C:lda &41

9A8E:cmp &34

9A90:bne &9A93

Exit

9A92:rts

Set the flags

9A93:ror A

9A94:eor &3B

9A96:rol A

Ensure Z=0

9A97:lda #&1

Exit

9A99:rts

Direct jump to the 'Type mismatch' error message

9A9A:jmp &8C0E

Alternative entry into comparison routine, which sets the flags according to the type first

9A9D:txa

Comparison routine

This routine compares two operands. On entry, the type of the first operand should be reflected in the flags. On exit, the following flags are in effect: (the two operands, of any type, are called A and B)

A = B Z = 1
A < > B Z = 0
A > B C = 1 and Z = 0
A < B C = 0

Goto &9AE7 if the first operand is a string

9A9E:beq &9AE7

Goto &9A50 if the first operand is real

9AA0:bmi &9A50

Integer comparison

Save the first operand on the stack

9AA2:jsr &BD94

Call LEVEL 4

9AA5:jsr &9C42

Examine the type of the second operand

9AA8:tay

Give a 'Type mismatch' error if it is a string

9AA9:beq &9A9A

Goto &9A39 if it is a real number

9AAB:bmi &9A39

Flip the sign of the second number. The sign bits of both numbers are flipped before subtraction takes place. This is to avoid the problem where a negative number will be interpreted as being bigger than any positive number by the SBC and CMP instructions

9AAD:lda &2D

9AAF:eor #&80

9AB1:sta &2D

Set the carry flag in preparation for the subtraction

9AB3:sec

Subtract the LSB's, leaving the result in the IAC

9AB4:ldy #&0

9AB6:lda (&4),Y

9AB8:sbc &2A

9ABA:sta &2A

Subtract the next bytes

9ABC:iny

9ABD:lda (&4),Y

9ABF:sbc &2B

9AC1:sta &2B

9AC3:iny

Subtract the next bytes

9AC4:lda (&4),Y

9AC6:sbc &2C

9AC8:sta &2C

Get the sign byte of the integer on the stack

9ACA:iny**9ACB:lda (&4),Y**

Set Y to zero for the 'TRUE'/'FALSE' computations done on exit

9ACD:ldy #&0

Flip the sign byte

9ACF:eor #&80

Subtract the MSB of the second number

9AD1:sub &2D

Check the rest of the number for zero

9AD3:ora &2A**9AD5:ora &2B****9AD7:ora &2C**

Save the flags

9AD9:php

Remove the integer from the stack

9ADA:clc**9ADB:lda #&4****9ADD:adc &4****9ADF:sta &4****9AE1:bcc &9AE5****9AE3:inc &5**

Get the flags back

9AE5:plp

Exit

9AE6:rts**String comparison**

Save the first string on the stack

9AE7:jsr &BDB2

Call LEVEL 4

9AEA:jsr &9C42

Give a 'Type mismatch' error if the second operand is not a string

9AED:tay**9AEE:bne &9A9A**

Save the next character

9AF0:stx &37

Get the length of the second string

9AF2:ldx &36

Get the length of the first string from the stack

9AF4:ldy #&0**9AF6:lda (&4),Y**

Save it in &39

9AF8:sta &39

Compare the lengths of the two strings

9AFA:cmp &36

If the second string is shorter than the first, skip the next instruction

9AFC:bcs &9AFF

Get the length of the shorter string into the X register

9AFE:tax

Save the length of the shorter string

9AFF:stx &3A

Goto &9B11 if the shorter string has a length of zero

9B01:ldy #&0

9B03:cpy &3A

9B05:beq &9B11

Point to the next character of the strings

9B07:iny

Compare a character from the second string to the corresponding character of the first string

9B08:lda (&4),Y

9B0A:cmp &5FF,Y

Continue if the characters are the same

9B0D:beq &9B03

Otherwise, exit

9B0F:bne &9B15

Compare the lengths of the strings

9B11:lda &39

9B13:cmp &36

Save the flags

9B15:php

Discard the string on the stack

9B16:jsr &BDDC

Get the next character back

9B19:ldx &37

Retrieve the flag settings

9B1B:plp

Exit

9B1C:rts

Evaluate expression at PTR #1

Copy PTR #1 to PTR #2

9B1D:lda &B

9B1F:sta &19

9B21:lda &C

9B23:sta &1A

9B25:lda &A

9B27:sta &1B

Fall into main expression evaluator

Evaluate expression at PTR #2

LEVEL 1

You will be familiar with the method used in the expression evaluator, from our earlier

discussions. It is easier to read the operand routine at &ADEC before starting to decipher this part of the routine. When each level exits, X contains the next character and A contains the type of the result of the level. Remember that the type identification bytes in the expression evaluator are different from those used in variable name scanning; &00 = string, &40 = integer and &FF = real

Call LEVEL 2

9B29:jsr &9B72

Check for 'OR'

9B2C:cpx #&84

9B2E:beq &9B3A

Check for 'EOR'

9B30:cpx #&82

9B32:beq &9B55

Decrement the offset of PTR #2, to go back over the character that is currently in X, since it is not part of the expression

9B34:dec &1B

Save the type in Y and &27

9B36:tay

9B37:sta &27

Exit

9B39:rts

OR operator routine

Save the result so far and call LEVEL 2

9B3A:jsr &9B6B

Set the flags according to the type of the second operand

9B3D:tay

Ensure it is an integer

9B3E:jsr &92F0

Point to the MSB of a four byte integer

9B41:ldy #&3

Get the current byte from the integer on the stack

9B43:lda (&4),Y

'OR' it with the corresponding byte in the IAC

9B45:ora &2A,Y

Save the result in the IAC

9B48:sta &2A,Y

Continue for all the bytes

9B4B:dey

9B4C:bpl &9B43

Discard the integer on top of the stack

9B4E:jsr &BDFF

Set the type to integer

9B51:lda #&40

Rejoin LEVEL 1

9B53:bne &9B2C

EOR operator routine

Save the result so far and call LEVEL 2

9B55:jsr &9B6B

Ensure the second operand is an integer

9B58:tay

9B59:jsr &92F0

Point to the third byte of the integers

9B5C:ldy #&3

Get the current byte from the first operand

9B5E:lda (&4),Y

Exclusive-OR it with the corresponding byte in the IAC

9B60:eor &2A,Y

Save the result in the IAC

9B63:sta &2A,Y

Continue this process until all the bytes have been dealt with

9B66:dey

9B67:bpl &9B5E

Join the code at the end of the 'OR' routine

9B69:bmi &9B4E

Ensure the current result is an integer

9B6B:tay

9B6C:jsr &92F0

Save it on the stack

9B6F:jsr &BD94

LEVEL 2

Call LEVEL 3

9B72:jsr &9B9C

Check for 'AND'

9B75:cpx #&80

9B77:beq &9B7A

Exit

9B79:rts

AND operator routine

Ensure the current operand is an integer

9B7A:tay

9B7B:jsr &92F0

Save it on the stack

9B7E:jsr &BD94

Call LEVEL 3

9B81:jsr &9B9C

Ensure the second operand is an integer

9B84:tay

9B85:jsr &92F0

Point to the fourth byte of the integers

9B88:ldy #&3

Get the current byte from the stack

9B8A:lda (&4),Y

AND it with the corresponding byte in the IAC

9B8C:and &2A,Y

Save the result in the IAC

9B8F:sta &2A,Y

Continue the process until all the bytes have been dealt with

9B92:dey

9B93:bpl &9B8A

Discard the integer on the top of the stack

9B95:jsr &BDFF

Set the result to integer

9B98:lda #&40

Rejoin LEVEL 2

9B9A:bne &9B75

LEVEL 3

Call LEVEL 4

9B9C:jsr &9C42

Exit if the next character is greater than or equal to &3F, which is the ASCII code for a '?'

9B9F:cpx #&3F

9BA1:bcs &9BA7

Jump to &9BA8 if the next character is greater than or equal to &3C, which is the ASCII code for '<'

9BA3:cpx #&3C

9BA5:bcs &9BA8

Exit

9BA7:rts

Deal with '<', '=' and '>'

Goto &9BC0 if the character is a '<'

9BA8:beq &9BC0

Goto &9BE8 if the character is '>'

9BAA:cpx #&3E

9BAC:beq &9BE8

= operator routine

Place the type of the first operand in X

9BAE:tax

Do the comparison

9BAF:jsr &9A9E

If the two operands are not equal, do not decrement Y, but leave it as 0

9BB2:bne &9BB5

If they are equal, decrement Y to &FF

9BB4:dey

Place Y in the IAC, giving either -1 or 0

9BB5:sty &2A

9BB7:sty &2B

9BB9:sty &2C

9BBB:sty &2D

Indicate that the result is an integer

9BBD:lda #&40

Exit

9BBF:rts

< operator routine

Save the type of the first operand in X

9BC0:tax

Get the next character

9BC1:ldy &1B

9BC3:lda (&19),Y

Goto &9BD4 if it is a '=', making the entire operator be '< ='

9BC5:cmp #&3D

9BC7:beq &9BD4

Goto &9BDF if it is a '>', making the entire operator be '< >'

9BC9:cmp #&3E

9BCB:beq &9BDF

Compare the operands

9BCD:jsr &9A9D

Return with 'TRUE' if the first was less than the second

9BD0:bcc &9BB4

Return with 'FALSE' otherwise

9BD2:bcs &9BB5

< = operator entry

Increment past the equals sign

9BD4:inc &1B

Compare the operands

9BD6:jsr &9A9D

Return with 'TRUE' if the first is less than or equal to the second

9BD9:beq &9BB4

9BDB:bcc &9BB4

Otherwise, return 'FALSE'

9BDD:bcs &9BB5

< > operator routine

Increment past the '>'

9BDF:inc &1B

Compare the operands

9BE1:jsr &9A9D

Return with 'TRUE' if the operands are not equal

9BE4:bne &9BB4

Return with 'FALSE' if the operands are equal

9BE6:beq &9BB5

> operator routine

Save the type of the first operand

9BE8:tax

Get the next character

9BE9:ldy &1B

9BEB:lda (&19),Y

If it is an equals sign, goto &9BFA

9BED:cmp #&3D

9BEF:beq &9BFA

Compare the operands

9BF1:jsr &9A9D

Return 'FALSE' if they are equal

9BF4:beq &9BB5

Return 'TRUE' if the first operand is greater than the second

9BF6:bcs &9BB4

Otherwise, return 'FALSE'

9BF8:bcc &9BB5

> = operator routine

Increment past the '='

9BFA:inc &1B

Compare the operands

9BFC:jsr &9A9D

Generate 'TRUE' if the second operand is less than the first

9BFF:bcs &9BB4

Otherwise, return 'FALSE'

9C01:bcc &9BB5

9C03:brk

9C04:13-

9C05:53-S

9C06:74-t

9C07:72-r

9C08:69-i

9C09:6E-n

9C0A:67-g

9C0B:20-

9C0C:74-t
9C0D:6F-o
9C0E:6F-o
9C0F:20-
9C10:6C-l
9C11:6F-o
9C12:6E-n
9C13:67-g
9C14:00-

String concatenation

The method used is to push the first string onto the stack before calling LEVEL 5 for the next operand. The second string is then moved up in the string buffer to make room for the first string, which is then downloaded from the stack

Save the first string on the stack

9C15:jsr &BDB2

Call LEVEL 5

9C18:jsr &9E20

Give a 'Type mismatch' error if the operand is not a string

9C1B:tay

9C1C:bne &9C88

Clear the carry flag in preparation for an ADC instruction

9C1E:clc

Save the next character

9C1F:stx &37

Get the length of the first string

9C21:ldy #&0

9C23:lda (&4),Y

Add it to the length of the second string

9C25:adc &36

Give a 'String too long' error if the combined length of the strings is greater than 255

9C27:bcs &9C03

Transfer the total length to X

9C29:tax

Save it on the machine stack

9C2A:pha

Get Y from the length of the second string

9C2B:ldy &36

Get a byte from the string

9C2D:lda &5FF,Y

Move it up

9C30:sta &5FF,X

Continue the process for all the characters in the string

9C33:dex

9C34:dey

9C35:bne &9C2D

Pull the first string from the stack

9C37:jsr &BDCB

Get the length of the combined string

9C3A:pla

Save this as the new string length

9C3B:sta &36

Get back the next character

9C3D:ldx &37

Set the type indicator to zero, which indicates a string result

9C3F:tya

Join the LEVEL 4 code

9C40:beq &9C45

LEVEL 4

Call LEVEL 5

9C42:jsr &9DD1

Check for '+'

9C45:cpx #&2B

9C47:beq &9C4E

Check for '-'

9C49:cpx #&2D

9C4B:beq &9CB5

Exit

9C4D:rts

+ operator routine

Inspect the type of the current operand

9C4E:tay

Do a string concatenation if the operand is a string

9C4F:beq &9C15

Do a floating point addition if the operand is real

9C51:bmi &9C8B

Integer addition

Push the current integer and call LEVEL 5

9C53:jsr &9DCE

Give a 'Type mismatch' error if the result is a string

9C56:tay

9C57:beq &9C88

Do a floating point addition if the result is a real number

9C59:bmi &9CA7

Start by adding the LSBs

9C5B:ldy #&0

Clear the carry flag in preparation for the addition

9C5D:clc

Add the LSBs

9C5E:lda (&4),Y

9C60:adc &2A

9C62:sta &2A

Add the next byte

9C64:iny

9C65:lda (&4),Y

9C67:adc &2B

9C69:sta &2B

Add the next byte

9C6B:iny

9C6C:lda (&4),Y

9C6E:adc &2C

9C70:sta &2C

Add the MSBs

9C72:iny

9C73:lda (&4),Y

9C75:adc &2D

9C77:sta &2D

Discard the integer on the stack

9C79:clc

9C7A:lda &4

9C7C:adc #&4

9C7E:sta &4

9C80:lda #&40

Rejoin the LEVEL 4 code

9C82:bcc &9C45

9C84:inc &5

Rejoin the LEVEL 4 code

9C86:bcs &9C45

Direct jump to the 'Type mismatch' error

9C88:jmp &8C0E

Floating point addition

Push the current number on the stack

9C8B:jsr &BD51

Call LEVEL 5

9C8E:jsr &9DD1

Give a 'Type mismatch' error if the result is a string

9C91:tay

9C92:beq &9C88

Save the next character

9C94:stx &27

Do not convert the result to a floating point number if it is real already

9C96:bmi &9C9B

Convert the integer in the IAC to a floating point number in FAC#1

9C98:jsr &A2BE

Discard the floating point number on the stack, leaving (&4B) pointing to the number

9C9B:jsr &BD7E

Add the floating point number pointed to by (&4B) to that in FAC#1

9C9E:jsr &A500

Get the next character back

9CA1:ldx &27

Indicate the result is real

9CA3:lda #&FF

Join the LEVEL 4 code

9CA5:bne &9C45

Save the next character

9CA7:stx &27

Pull the integer off the stack

9CA9:jsr &BDEA

Save FAC#1 on the stack

9CAC:jsr &BD51

Convert the integer in the IAC to a real number in FAC#1

9CAF:jsr &A2BE

Join the normal addition code

9CB2:jmp &9C9B

- operator routine

Give a 'Type mismatch' error if the operand is a string

9CB5:tay

9CB6:beq &9C88

Go and do a floating point subtraction if the operand is real

9CB8:bmi &9CE1

Integer subtraction

Save the current operand and call LEVEL 5

9CBA:jsr &9DCE

Give a 'Type mismatch' error if the result is a string

9CBD:tay

9CBE:beq &9C88

Go and do a floating point subtraction if the result is a real number

9CC0:bmi &9CFA

Set the carry flag in preparation for the subtraction

9CC2:sec

Subtract the LSBs

9CC3:ldy #&0

9CC5:lda (&4),Y

9CC7:sbc &2A**9CC9:sta &2A**

Subtract the next bytes

9CCB:iny**9CCC:lda (&4),Y****9CCE:sbc &2B****9CD0:sta &2B**

Subtract the next bytes

9CD2:iny**9CD3:lda (&4),Y****9CD5:sbc &2C****9CD7:sta &2C**

Subtract the MSBs

9CD9:iny**9CDA:lda (&4),Y****9CDC:sbc &2D**

Join the code at the end of the integer addition routine

9CDE:jmp &9C77**Floating point subtraction**

Push the first operand on the stack

9CE1:jsr &BD51

Call LEVEL 5

9CE4:jsr &9DD1

Give a 'Type mismatch' error if the result is a string

9CE7:tay**9CE8:beq &9C88**

Save the next character

9CEA:stx &27

Convert it to a floating point number if it is an integer

9CEC:bmi &9CF1**9CEE:jsr &A2BE**

Discard the floating point number on the stack, but leave (&4B) pointing to it

9CF1:jsr &BD7E

Set FAC#1 = (&4B) - FAC#1

9CF4:jsr &A4FD

Join the code at the end of the floating point subtraction routine

9CF7:jmp &9CA1

Save the next character

9CFA:stx &27

Pull the integer operand from the stack

9CFC:jsr &BDEA

Push FAC#1

9CFF:jsr &BD51

Convert the integer operand to a floating point number

9D02:jsr &A2BE

Discard the floating point number at the top of the stack, leaving (&4B) pointing to it

9D05:jsr &BD7E

Set $FAC\#1 = (\&4B) - FAC\#1$

9D08:jsr &A4D0

Join the code at the end of the flag

Floating point subtraction

Push the first operand on the stack

9CE1:jsr &BD51

Call LEVEL 5

9CE4:jsr &9DD1

Give a 'Type mismatch' error if the result is a string

9CE7:tay**9CE8:beq &9C88**

Save the next character

9CEA:stx &27

Convert it to a floating point number if it is an integer

9CEC:bmi &9CF1**9CEE:jsr &A2BE**

Discard the floating point number on the stack, but leave (&4B) pointing to it

9CF1:jsr &BD7E

Set $FAC\#1 = (\&4B) - FAC\#1$

9CF4:jsr &A4FD

Join the code at the end of the floating point subtraction routine

9CF7:jmp &9CA1

Save the next character

9CFA:stx &27

Pull the integer operand from the stack

9CFC:jsr &BDEA

Push $FAC\#1$

9CFF:jsr &BD51

Convert the integer operand to a floating point number

9D02:jsr &A2BE

Discard the floating point number at the top of the stack, leaving (&4B) pointing to it

9D05:jsr &BD7E

Set $FAC\#1 = (\&4B) - FAC\#1$

9D08:jsr &A4D0

Join the code at the end of the floating point addition routine

9D0B:jmp &9CA1

Convert the integer in the IAC to a floating point number in $FAC\#1$

9D0E:jsr &A2BE

Pull the integer off the stack

9D11:jsr &BDEA

Push the floating point number on the stack

9D14:jsr &BD51

Convert the integer to a real number

9D17:jsr &A2BE

Join the floating point multiplication routine

9D1A:jmp &9D2C

Convert the first operand to a real number, then fall into floating point multiplication

9D1D:jsr &A2BE

Floating point multiplication

Save the operand on the stack

9D20:jsr &BD51

Call LEVEL 6

9D23:jsr &9E20

Save the next character

9D26:stx &27

Ensure the second operand is a real number

9D28:tay

9D29:jsr &92FD

Discard the operand at the top of the stack, leaving (&4B) pointing to the result

9D2C:jsr &BD7E

Call the multiplication routine

9D2F:jsr &A656

Indicate a floating point result

9D32:lda #&FF

Retrieve the next character

9D34:ldx &27

Rejoin LEVEL 5

9D36:jmp &9DD4

Direct jump to the 'Type mismatch' error

9D39:jmp &8C0E

*** operator routine**

Give a 'Type mismatch' error if the operand is a string

9D3C:tay

9D3D:beq &9D39

Do a floating point multiplication if the operand is real

9D3F:bmi &9D20

Integer multiplication

The integer multiplication algorithm is complicated by the checks that are made for overflow before it starts. The problem is that a line such as 'PRINT &100000*&100000', if carried out wholly in integer arithmetic, would cause overflow. Thus, the multiplication routine tries to sense whether a multiplication would be better carried out using floating point arithmetic before proceeding with the sum. This

is done by checking if either operand is outside the range -&8000 to &7FFF. If either is, floating point multiplication is used. This system is a little conservative. The range checking is carried out in a rather opaque way

Do real multiplication if the top two bytes of the operand are not equal

9D41:lda &2D

9D43:cmp &2C

9D45:bne &9D1D

Skip to &9DFE if the top byte is zero - this implies both bytes are zero

9D47:tay

9D48:beq &9D4E

If the byte is not &FF, go back to do a floating point multiplication

9D4A:cmp #&FF

9D4C:bne &9D1D

By the time we get here, we can be certain that the top two bytes of the operand are either &0000 or &FFFF. This puts the number in the range -&10000 to &FFFF. The next step is to make sure that the top bit of the second byte of the number is the same as the top bit of the MSB of the number

9D4E:eor &2B

9D50:bmi &9D1D

Save the current operand and call LEVEL 6

9D52:jsr &9E1D

Save the next character

9D55:stx &27

Give a 'Type mismatch' error if the result is a string

9D57:tay

9D58:beq &9D39

Go back to do a floating point multiplication if the result is real

9D5A:bmi &9D11

Carry out the same checks on the range of the number

9D5C:lda &2D

9D5E:cmp &2C

9D60:bne &9D0E

9D62:tay

9D63:beq &9D69

9D65:cmp #&FF

9D67:bne &9D0E

9D69:eor &2B

9D6B:bmi &9D0E

Save the sign byte of the number

9D6D:lda &2D

9D6F:pha

Take the absolute value of the number

9D70:jsr &AD71

Copy the IAC to &39 onwards

9D73:ldx #&39

9D75:jsr &BE44

Pull the first operand

9D78:jsr &BDEA

Retrieve the other sign

9D7B:pla

Exclusive-OR it with the current sign to get the sign of the answer

9D7C:eor &2D

Save the sign of the answer in &37

9D7E:sta &37

Take the absolute value of the number

9D80:jsr &AD71

The algorithm used for multiplication is similar to the one we studied earlier. If the numbers to be multiplied are in 'X' and 'Y' (which correspond to locations &39,&3A and the IAC respectively) and the answer will be formed in 'A' (which is registers X and Y and locations &3F and &40), the algorithm can be expressed as follows:

1.A = 0

2.REPEAT

3.LSR X

4.BCC 6

5.A = A + Y

6.ASL Y

7.UNTIL X = 0

The principle of the routine is identical to the one we studied. Notice the way that maximum use is made of the X and Y registers to speed the routine up

Zero 'A'

9D83:ldy #&0

9D85:ldx #&0

9D87:sty &3F

9D89:sty &40

Shift 'X' right

9D8B:lsr &3A

9D8D:ror &39

Do not do the addition if there is no carry

9D8F:bcc &9DA6

Set 'A' = 'A' + 'Y'

9D91:clc

9D92:tya

9D93:adc &2A

9D95:tay

9D96:txa

9D97:adc &2B

9D99:tax

9D9A:lda &3F

9D9C:adc &2C

9D9E:sta &3F

9DA0:lda &40

9DA2:adc &2D

9DA4:sta &40

Shift 'Y' left

9DA6:asl &2A

9DA8:rol &2B

9DAA:rol &2C

9DAC:rol &2D

Continue until 'X' is zero

9DAE:lda &39

9DB0:ora &3A

9DB2:bne &9D8B

Save the less significant two bytes of the result

9DB4:sty &3D

9DB6:stx &3E

Save the sign of the result

9DB8:lda &37

9DBA:php

Copy the result, at &3D onwards, to the IAC

9DBB:ldx #&3D

9DBD:jsr &AF56

Get the sign back

9DC0:plp

Negate the answer if it should be positive

9DC1:bpl &9DC6

9DC3:jsr &AD93

Get back the next letter

9DC6:ldx &27

Join the LEVEL 5 code

9DC8:jmp &9DD4

Jump to join the multiplication routine

9DCB:jmp &9D3C

Save the current integer operand

9DCE:jsr &BD94

LEVEL 5

Call LEVEL 6

9DD1:jsr &9E20

Check for multiplication

9DD4:cpx #&2A

9DD6:beq &9DCB

Check for division

9DD8:cpx #&2F

9DDA:beq &9DE5

Check for 'MOD'

9DDC:cpx #&83

9DDE:beq &9E01

Check for 'DIV'

9DE0:cpx #&81

9DE2:beq &9E0A

Exit

9DE4:rts

/ operator routine

Ensure the first operand is real

9DE5:tay

9DE6:jsr &92FD

Save it on the stack

9DE9:jsr &BD51

Call LEVEL 6

9DEC:jsr &9E20

Save the next character

9DEF:stx &27

Ensure the divisor is real

9DF1:tay

9DF2:jsr &92FD

Discard the dividend at the top of the stack, leaving (&4B) pointing to it

9DF5:jsr &BD7E

Set $FAC \# 1 = (&4B) / FAC \# 1$

9DF8:jsr &A6AD

Get the next character back

9DFB:ldx &27

Indicate that the result is real

9DFD:lda #&FF

Rejoin the LEVEL 5 code

9DFF:bne &9DD4

MOD operator routine

Call the integer division routine

9E01:jsr &99BE

Save the sign of the dividend

9E04:lda &38

9E06:php

Join the code at the end of the integer multiplication routine

9E07:jmp &9DBB

DIV operator routine

Call the integer division routine

9E0A:jsr &99BE

Multiply the dividend, which contains the quotient, by two to allow the final iteration to take place

9E0D:rol &39

9E0F:rol &3A

9E11:rol &3B

9E13:rol &3C

Save the sign of the result

9E15:bit &37

9E17:php

Copy the quotient into the IAC and exit via the end of the integer multiplication routine

9E18:ldx #&39

9E1A:jmp &9DBD

Save the current integer operand on the stack

9E1D:jsr &BD94

LEVEL 6

Call the operand routine

9E20:jsr &ADEC

Save the type of the operand

9E23:pha

Skip spaces before the next operand

9E24:ldy &1B

9E26:inc &1B

9E28:lda (&19),Y

9E2A:cmp #&20

9E2C:beq &9E24

Transfer the next character to the X register

9E2E:tax

Get the type back

9E2F:pla

Check for exponentiation

9E30:cpx #&5E

9E32:beq &9E35

Exit

9E34:rts

Exponentiation operator

This routine is the first complex floating point routine. It evaluates exponentiation operations, as in 'A ^B'

The method used is quite devious. If B is an integer, repeated multiplication is used to get the answer. If B is not an integer, the answer is worked out from $(A \wedge \text{INT}(B)) * (A \wedge \text{FLOAT}(B))$, where FLOAT is a non-existent (in BBC BASIC) function

that returns the fractional part of a number. There are some added complications

because of the range of values for B the repeated multiplication algorithm will accept. The floating point exponentiation routine, used for the decimal part of B, simply calls the LN and EXP routines in the established way. (Using the formula $R^H = \text{EXP}(H \cdot \text{LN}(R))$, which is a natural consequence of logarithms)

Ensure that A is real

9E35:tay

9E36:jsr &92FD

Save A on the stack

9E39:jsr &BD51

Get B

9E3C:jsr &92FA

Goto &9E88 if the exponent of B is greater than or equal to &87. This means a branch will occur when B is greater than or equal to 64

9E3F:lda &30

9E41:cmp #&87

9E43:bcs &9E88

Get the fractional part of B into FAC#1, with the integer part in &4A. The integer part is expressed as a two's complement byte

9E45:jsr &A486

If B has a fractional part, branch to &9E59. Thus, control only reaches &9E4A if B is a simple integer between -64 and +64

9E48:bne &9E59

Discard A from the stack, leaving (&4B) pointing to it

9E4A:jsr &BD7E

Unpack A to FAC#1

9E4D:jsr &A3B5

Get B into the accumulator

9E50:lda &4A

Let FAC#1 = FAC#1 accumulator

9E52:jsr &AB12

Indicate that the result is real

9E55:lda #&FF

Rejoin the LEVEL 6 code

9E57:bne &9E23

Control passes here when B is not an integer

Pack the fractional part of B to &476 onwards

9E59:jsr &A381

Make (&4B) point to A on the stack

9E5C:lda &4

9E5E:sta &4B

9E60:lda &5

9E62:sta &4C

Unpack A to FAC#1

9E64:jsr &A3B5

Let $\text{FAC}\#1 = \text{FAC}\#1$ (integer part of B)

9E67:lda &4A

9E69:jsr &AB12

Save the answer at &471 onwards

9E6C:jsr &A37D

Discard A from the stack, leaving (&4B) pointing to it

9E6F:jsr &BD7E

Unpack A from (&4B) to $\text{FAC}\#1$

9E72:jsr &A3B5

Set $\text{FAC}\#1 = \text{LN}(A)$

9E75:jsr &A801

Set $\text{FAC}\#1 = \text{FAC}\#1 * B$

9E78:jsr &AAD1

Set $\text{FAC}\#1 = \text{EXP}(\text{FAC}\#1)$

9E7B:jsr &AA94

Make (&4B) point to &471, which contains the first partial answer

9E7E:jsr &A7ED

Multiply the two partial answers together to get the final answer

9E81:jsr &A656

Indicate that the result is real and exit

9E84:lda #&FF

9E86:bne &9E23

Pack B to &471

9E88:jsr &A381

Set $\text{FAC}\#1$ to 1

9E8B:jsr &A699

Join the previous code

9E8E:bne &9E6C

Print a number in hexadecimal

Notice the cunning way in which leading zeros are suppressed

Ensure that the number is an integer

9E90:tya

9E91:bpl &9E96

9E93:jsr &A3E4

Zero the pointer into the digit store

9E96:ldx #&0

Zero the pointer into the IAC

9E98:ldy #&0

Get the next byte from the IAC

9E9A:lda &2A,Y

Save it on the stack

9E9D:pha

Isolate the right hand nybble

```

    9E9E:and  #&F
Save it in the digit store
    9EA0:sta  &3F,X
Retrieve the byte
    9EA2:pla
Isolate the left hand nybble
    9EA3:lsr  A
    9EA4:lsr  A
    9EA5:lsr  A
    9EA6:lsr  A
Save it in the next digit store location
    9EA7:inx
    9EA8:sta  &3F,X
Point to the next digit store location
    9EAA:inx
Increment the pointer into the IAC
    9EAB:iny
Continue until it reaches 4
    9EAC:cpy  #&4
    9EAE:bne  &9E9A
Point to the last digit store position
    9EB0:dex
Exit if the end of the digit store has been reached
    9EB1:beq  &9EB7
Get the next digit
    9EB3:lda  &3F,X
If it is zero, loop backwards. Thus when the digit store is scanned, any leading zero
digits will be ignored
    9EB5:beq  &9EB0
Get the digit from the digit store
    9EB7:lda  &3F,X
Add six to it if it is not a decimal digit
    9EB9:cmp  #&A
    9EBB:bcc  &9EBF
    9EBD:adc  #&6
Add the ASCII factor
    9EBF:adc  #&30
Add the digit to the string buffer
    9EC1:jsr  &A066
Continue the process until the end of the digit store
    9EC4:dex
    9EC5:bpl  &9EB7
Exit
    9EC7:rts
Goto &9ED1 if the number is positive

```

9EC8:bpl &9ED1

Reset the sign bit, to make the number positive

9ECA:lda #&2D**9ECC:sta &2E**

Add a minus sign to the output string

9ECE:jsr &A066

Goto &9F25 if the exponent of the number is greater than or equal to &81. The exponent will only be greater than or equal to &81 if the number itself is one or more

9ED1:lda &30**9ED3:cmp #&81****9ED5:bcs &9F25**

Multiply the number by 10

9ED7:jsr &A1F4

Decrement EXPCNT to indicate the decimal point has shifted one position to the left

9EDA:dec &49

Repeat the process

9EDC:jmp &9ED1**Convert a number to a string**

This routine ranks as one of the most complex and difficult to understand in the entire ROM. To circumvent this problem, I will attempt to explain it in two stages. The first stage is to give a BASIC program (which you can actually run) which is equivalent to the routine and the second is to look at the routine itself.

First, familiarise yourself with the action of @% in the PRINT statement. This is covered adequately in the User Guide on pages 326/7.

This is the BASIC version of the routine:

> LIST

```

10 REM Filename:PRNTDEM
20
30 REM BASIC II PRINT routine
40
50 REM (c) 1983 Jeremy Ruston & Acorn
60
70 REM -----
80 A = 0
90 F% = 2
100 D% = 0
110 P4E% = 2
120 P38% = 2
130 S$ = ""
140 REM -----
150 INPUT "Enter number:" A
160 IF A = 0 THEN STOP

```



```

170 REM -----
180 GOSUB 220
190 REM -----
200 PRINT S$
210 END
220 REM -----
230 IF SGN(A) = -1 THEN S$ = S$ + "-":A = -A
240 REM -----
250 IF A < 1 THEN A = A*10:D% = D%-1:GOTO 250
260 IF A > = 10 THEN A = A/10:D% = D% + 1:GOTO 260
270 REM -----
280 TEMP = A
290 REM -----
300 P38% = P4E%
310 IF F% < > 2 THEN GOTO 390
320 T% = D% + P38% + 1
330 IF SGN(T%) = -1 THEN GOTO 590
340 P38% = T%
350 IF T% < 11 THEN GOTO 390
360 P38% = 10
370 F% = 0
380 REM -----
390 A = 5
400 X% = P38%
410 IF X% = 0 THEN GOTO 460
420 A = A/10
430 X% = X%-1
440 IF X% < > 0 THEN GOTO 420
450 REM -----
460 A = A + TEMP
470 REM -----
480 E% = ?(TOP + 3)
490 M% = (?(TOP + 4) OR &80)*&10000 + ?(TOP + 5)*&100 + ?(
TOP + 6)
500 REM -----
510 IF E% > = &84 THEN GOTO 560
520 M% = M% DIV 2
530 E% = E% + 1
540 GOTO 510
550 REM -----
560 IF ?&436 > = &A0 THEN A = 1:D% = D% + 1:GOTO 250
570 REM -----
580 IF P38% < > 0 THEN GOTO 630
590 A = 0
600 D% = 0

```

```

610 P38% = P4E% + 1
620 REM -----
630 IF F% = 1 THEN A% = 1:GOTO 790
640 IF SGN(D%) = -1 THEN GOTO 690
650 IF D% > = P38% THEN A% = 1:GOTO 790
660 A% = D% + 1:D% = 0
670 GOTO 790
680 REM -----
690 IF F% = 2 THEN GOTO 720
700 A% = 1
710 IF D% < > -1 THEN GOTO 790
720 S$ = S$ + "0."
730 D% = D% + 1
740 IF D% = 0 THEN GOTO 770
750 S$ = S$ + "0"
760 GOTO 730
770 A% = &80
780 REM -----
790 P4E% = A%
800 GOSUB 960
810 P4E% = P4E%-1
820 IF P4E% = 0 THEN S$ = S$ + "."
830 P38% = P38%-1
840 IF P38% < > 0 THEN GOTO 800
850 REM -----
860 IF F% = 1 THEN GOTO 930
870 IF F% = 2 THEN GOTO 920
880 REM -----
890 IF RIGHT$(S$,1) = "0" THEN S$ = LEFT$(S$,LEN(S$)-1):GOTO
890
900 IF RIGHT$(S$,1) = "." THEN S$ = LEFT$(S$,LEN(S$)-1)
910 REM -----
920 IF D% = 0 THEN RETURN
930 S$ = S$ + "E" + STR$(D%)
940 RETURN
950 REM -----
960 S$ = S$ + CHR$( (?&436 DIV 16) + &30 )
970 ?&436 = ?&436 AND &F
980 M% = M%*10
990 RETURN

```

>

The horizontal lines are meant to break up the routine.

Lines 80 to 130 initialise the variables used by the program. 'A' is initialised first. This is the variable that will be printed. Because it is the first variable to be defined, it can

easily be accessed by the byte indirection operator, which allows the program to directly manipulate the exponent and mantissa of the number. 'F%' gives the format in which the number will be printed. It thus corresponds to B3 of @%. 'D%' is used to keep track of the decimal exponent of the number. This is the number printed after the 'E'. 'P4E%' and 'P38%' are both set to the number of digits to be printed. They thus correspond to B2 of @%. 'S\$' is the string variable used to hold the number as it is converted.

Lines 150 and 160 accept the number to be printed from the user and ensure the number is not zero. The assembly language version of the routine includes code to deal with the case of the number being zero, but this would merely confuse the BASIC program.

Line 180 calls the routine. Notice that I have coded the routine to use GOSUBs and GOTOs. While this is terrible practice, it does emphasise the connection between the BASIC program and the assembly language.

Lines 200 and 210 print the string corresponding to the number and terminate the program.

Line 230 starts the routine by checking if 'A' is negative. If it is, a minus sign is put into the string and the number is negated.

Lines 250 and 260 make sure 'A' is between 1 and 10 (in fact, they ensure $1 < = A < 10$). In line 250, the routine checks whether the number is less than 1. If it is, the number is multiplied by 10, and the decimal exponent is decremented. The test is then repeated. Line 260 carries out the same procedure for ensuring the number is less than 10. The number is now of the form:

0000000A.BCDEFGH

The decimal exponent, 'D%', tells us where the decimal point was originally. If 'D%' is negative, the decimal point was to the left of its current position. For example, if 'D%' were given as -3, the number would have originally been:

00000.00ABCDEFGH

Similarly, if 'D%' were given as +5, the number would have been

0000000ABCDEF.GH

Line 280 saves the number in 'TEMP'.

Lines 300 to 370 adjust 'P38%' so that it reflects the number of digits to be actually printed. This is because 'P38%' only gives the number of digits to be printed after the decimal point in 'F' mode. Line 300 updates 'P38%' from the number of digits specified. Line 310 skips the routine if 'F' mode is not being used. Line 320 computes the total number of digits from the number after the decimal point, plus the number before plus one. Line 330 tests for a negative result. If the result is negative (as in 'Print 0.0000686 with three digits of accuracy'), zero must be printed, so control passes to line 590. Line 340 assigns the total number of digits to 'P38%'. At this point, 'P38%'

will contain a different number to 'P4E%', which is why two variables are used. Lines 350 to 370 ensure that the total is less than 11. If the total is greater, 'G' mode is used. Lines 390 to 440 compute a rounding factor to add on to the number. The normal way of rounding a number is as follows:

If the number A.BCDEFGH is to be rounded to 7 digits rather than eight, digit 'H' should be examined. If it is less than five, the number can be rounded by simply ignoring the 'H' digit. If 'H' is five or greater, digit 'G' must be incremented. (This incrementation can ripple throughout the rest of the digits of the number). In mathematical terms, 0.0000005 must be added onto the original number. Thus, this section of code simply computes the number that should be added to the number to be printed in order to round it. Line 390 sets the factor to 5. Line 400 moves 'P38%' into a convenient register so that it can be used as the index to a loop. Line 410 skips the rest of the routine if the 'X%' is zero. If 'X%' is zero on entry, no digits are to be printed, so 5 is the rounding factor. Otherwise, line 420 divides the rounding factor by 10, to move the decimal point one position to the left. Lines 430 and 440 then check for the end of the loop, repeating the process if the loop has yet to complete.

Line 460 adds the rounding factor onto the number to be printed.

Lines 480 and 490 extract the two parts of the number from the variable storage area. This is done to allow the exponent and mantissa to be manipulated separately. Notice that the '!' operator cannot be used to access the mantissa, since the mantissa is stored from MSB to LSB.

Lines 510 to 540 attempt to manipulate the mantissa and exponent until the bicimal point is positioned four bits from the left of the number. This will leave the integer part of the number in the top four bits of the mantissa. Line 510 leaves the routine if the exponent already indicates that the bicimal point is in the correct position. Line 520 shifts the mantissa one position to the right, while line 530 increments the exponent to compensate. Line 540 then passes control back to the start of this section.

Line 560 checks whether the integral part of the number is greater than 9. If it is, the rounding process produced a carry out of the top digit. It carries out the check by inspecting the top bits of the mantissa. If they are greater than 9, the number to be printed is set to 1 and the decimal exponent incremented before control is passed right back to the start of the subroutine.

Lines 580 to 610 set the number to be printed to zero if no digits are to be printed.

Line 630 checks whether 'E' format is in use. If it is, the accumulator (represented by A%) is set to 1. This indicates to the next part of the routine that the decimal point should be printed after the first digit of the number. Control is then passed to line 790 which starts the routine to actually print the number.

Line 640 checks for the exponent being negative. If it is, the routine at 690 is executed. This is to allow the computer to choose whether to use exponential format for fractions, or whether to print them directly.

Line 650 checks whether the number can be printed without using 'E' mode with the current digits setting. If the number cannot be printed except in 'E' mode, the system is told to print the decimal point after the first digit and control passes to line 790.

Line 660 tells the system to print the decimal point at its normal position. The decimal exponent is set to zero to prevent it being printed.

Line 670 jumps to skip the code concerned with fractions.

Line 690 skips the next two lines if in 'F' mode.

Lines 700 and 710 use 'E' mode whilst in 'G' format if the number is smaller than 0.1.

Line 720 adds the first part of the fraction to the output string.

Line 730 increments the decimal exponent.

Line 740 leaves the routine if the exponent has become zero, since that indicates that the first digit should be printed.

Line 750 adds a zero to the string.

Line 760 repeats the process.

Line 770 sets the position of the decimal point to &80 to ensure that it is not printed.

Line 790 saves the position of the decimal point.

Line 800 adds the next digit to the string buffer.

Lines 810 and 820 include a decimal point if necessary.

Lines 830 and 840 loop backwards for the remaining digits.

Line 860 jumps forwards to print the exponent if 'E' mode is in use.

Line 870 jumps forwards if in 'F' mode and prints the exponent as long as it is not zero.

Line 890 removes the trailing zeros from the end of the 'G' format number.

Line 900 removes the single trailing decimal point if it is present.

Line 920 decides not to print the exponent if it is zero.

Line 930 is a bit of a cheat, but it adds the exponent to the end of the buffer.

Lines 960 to 990 make up the subroutine that adds the next digit of the number to the string buffer. Line 960 derives a digit from the most significant nybble of the mantissa and adds it onto the end of the string buffer. Line 970 removes the digit from the mantissa. Line 980 moves the mantissa one decimal place to the left, using an integer multiply routine. Line 990 then exits the routine. This short routine is considerably easier to understand when the assembly language version is studied.

It is difficult to gain a good understanding of the routine without running the above program. It is instructive to alter the routine, or see how it functions with lines added or removed

Store the format in &37. If the format given is invalid (three or over), use 'G' format

9EDF:ldx &402

9EE2:cpx #&3

9EE4:bcc &9EE8

9EE6:ldx #&0

9EE8:stx &37

Get the number of digits to be printed

9EEA:lda &401

Goto &9EF5 if no digits are specified

9EED:beq &9EF5

Use ten digits if more than ten digits are specified

9EEF:cmp #&A

9EF1:bcs &9EF9

Otherwise use the number specified

9EF3:bcc &9EFB

Only allow zero digits to be used in 'F' format

9EF5:cpx #&2

9EF7:beq &9EFB

Get the number of digits to be printed

9EF9:lda #&A

Save the number of digits to be printed in &38 and &4E

9EFB:sta &38

9EFD:sta &4E

Zero the length of the string in the buffer

9EFF:lda #&0

9F01:sta &36

Zero EXPCNT

9F03:sta &49

Go back to do a hexadecimal conversion if the hex flag is set

9F05:bit &15

9F07:bmi &9E90

Ensure the number to be printed is real

9F09:tya

9F0A:bmi &9F0F

9F0C:jsr &A2BE

Goto &9EC8 if the number is not zero

9F0F:jsr &A1DA

9F12:bne &9EC8

Goto &9F1D if 'G' format is not in use

9F14:lda &37

9F16:bne &9F1D

Exit, with zero in the string buffer

9F18:lda #&30

9F1A:jmp &A066

Print the number

9F1D: jmp &9F9C

Set FAC#1 = 1

9F20: jsr &A699

Join the main code

9F23: bne &9F34

Goto &9F39 if the number is less than eight

9F25: cmp #&84

9F27: bcc &9F39

Goto &9F31 if the number is greater than 16

9F29: bne &9F31

Goto &9F39 if the number is less than ten. Remember that if control reaches here the number has an exponent of &84 and that in floating point notation, ten has an exponent of &84 and a mantissa of &A0000000

9F2B: lda &31

9F2D: cmp #&A0

9F2F: bcc &9F39

Divide the number by 10

9F31: jsr &A24D

Increment EXPCNT to indicate that the decimal point has been moved one position to the left

9F34: inc &49

Repeat the process

9F36: jmp &9ED1

Save the number at &46C onwards. Notice that the normal packing routine does not save the rounding byte, so this is saved separately

9F39: lda &35

9F3B: sta &27

9F3D: jsr &A385

Update the number of digits to be printed from the number of digits specified. This is necessary when the control passes back to the beginning of the routine

9F40: lda &4E

9F42: sta &38

If 'F' format is being used, the number of digits specified only specifies the number of digits after the decimal point, so the number of digits before the decimal point has to be added. Before this can be done, the routine is skipped if 'F' format is not being used

9F44: ldx &37

9F46: cpx #&2

9F48: bne &9F5C

Add the number of digits after the decimal point to the decimal exponent. In view of the fact that the carry flag will always be set at this point, this instruction computes in A the total number of digits to be printed

9F4A: adc &49

If the result is negative, the number specified results in zero being printed

9F4C:bmi &9FA0

Save the number of digits

9F4E:sta &38

If the number of digits is greater than 10, indicate ten digits are to be printed and use 'G' format

9F50:cmp #&B

9F52:bcc &9F5C

9F54:lda #&A

9F56:sta &38

9F58:lda #&0

9F5A:sta &37

Set FAC#1 to 0

9F5C:jsr &A686

Change FAC#1 to 5 by changing the exponent and the most significant byte of the mantissa

9F5F:lda #&A0

9F61:sta &31

9F63:lda #&83

9F65:sta &30

Get the number of digits remaining into X

9F67:ldx &38

If zero digits remain, exit the loop

9F69:beq &9F71

Divide FAC#1 by 10

9F6B:jsr &A24D

Continue until the digit count is zero

9F6E:dex

9F6F:bne &9F6B

Retrieve FAC#2 from &46C

9F71:jsr &A7F5

9F74:jsr &A34E

9F77:lda &27

9F79:sta &42

Add FAC#2 to FAC#1

9F7B:jsr &A50B

Exit the loop if the exponent is greater than &84

9F7E:lda &30

9F80:cmp #&84

9F82:bcs &9F92

Shift the mantissa right

9F84:ror &31

9F86:ror &32

9F88:ror &33

```

9F8A:ror &34
9F8C:ror &35
Increment the exponent to compensate
9F8E:inc &30
Continue
9F90:bne &9F7E

Return to the start if the integral part is 10 or greater
9F92:lda &31
9F94:cmp #&A0
9F96:bcs &9F20

Skip this section if the number of digits to be printed is not zero
9F98:lda &38
9F9A:bne &9FAD
Goto &9FE6 if 'E' format is in use (these two instructions are only used when control
passes here from &9F1D)
9F9C:cmp #&1
9F9E:beq &9FE6
Set FAC#1 to zero
9FA0:jsr &A686
Zero the decimal exponent
9FA3:lda #&0
9FA5:sta &49
Set the number of digits to be printed to the number specified in @%
9FA7:lda &4E
9FA9:sta &38
Increment the result to allow for the leading zero
9FAB:inc &38

Goto &9FE6 to print the number directly if 'E' format is in use
9FAD:lda #&1
9FAF:cmp &37
9FB1:beq &9FE6
Print the necessary leading zero's if the number is less than one
9FB3:ldy &49
9FB5:bmi &9FC3

If the number will fit into the specified number of digits, go on to print it
9FB7:cpy &38
9FB9:bcs &9FE6
Otherwise, zero the exponent
9FBB:lda #&0
9FBD:sta &49
And include the decimal point (which is indicated in 'A') at its normal position
9FBF:iny
9FC0:tya

```


Jump to print the number

9FC1:bne &9FE6

If 'F' format is not being used, only print the leading zeros if the number is greater than or equal to 0.1

9FC3:lda &37

9FC5:cmp #&2

9FC7:beq &9FCF

9FC9:lda #&1

9FCB:cpy #&FF

9FCD:bne &9FE6

Add an ASCII zero to the string

9FCF:lda #&30

9FD1:jsr &A066

Add a decimal point to the string

9FD4:lda #&2E

9FD6:jsr &A066

Prime the accumulator with an ASCII zero

9FD9:lda #&30

Increment the exponent and exit if it becomes zero

9FDB:inc &49

9FDD:beq &9FE4

Add the zero to the buffer

9FDF:jsr &A066

Repeat the process

9FE2:bne &9FDB

Indicate that the decimal point is at &80, which will prevent it being printed

9FE4:lda #&80

Save the location of the decimal point

9FE6:sta &4E

Add the next digit to the buffer

9FE8:jsr &A040

Decrement the decimal point counter

9FEB:dec &4E

Add a decimal point to the buffer if it reaches zero

9FED:bne &9FF4

9FEF:lda #&2E

9FF1:jsr &A066

Continue the process for the required number of digits

9FF4:dec &38

9FF6:bne &9FE8

Goto &A015 to print the exponent part of the number if 'E' mode is in use

9FF8:ldy &37

9FFA:dey

9FFB:beq &A015

If 'F' mode is in use, goto &A011 and print the exponent as long as it is not zero

9FFD:dey

9FFE:beq &A011

For 'G' format numbers, remove the trailing zeros from the end of the buffer

A000:ldy &36

A002:dey

A003:lda &600,Y

A006:cmp #&30

A008:beq &A002

And the decimal point if needed

A00A:cmp #&2E

A00C:beq &A00F

A00E:iny

A00F:sty &36

Return if the exponent is zero

A011:lda &49

A013:beq &A03F

Add an 'E' to the buffer

A015:lda #&45

A017:jsr &A066

Skip to &A028 if the exponent is positive

A01A:lda &49

A01C:bpl &A028

Add a minus sign to the buffer

A01E:lda #&2D

A020:jsr &A066

Negate the exponent

A023:sec

A024:lda #&0

A026:sbc &49

Add the exponent to the buffer

A028:jsr &A052

Exit if 'G' format is in use

A02B:lda &37

A02D:beq &A03F

Prime the accumulator with an ASCII space

A02F:lda #&20

Get the exponent into Y

A031:ldy &49

Add a space to the buffer (to make up for the minus sign) if the exponent is positive

A033:bmi &A038

A035:jsr &A066

Add another space if the exponent is a single digit

A038:cpx #&0

A03A:bne &A03F**A03C:jmp &A066**

Exit

A03F:rts**Add next digit of MAN#1 to buffer**

This routine is used by the decimal printing routine to add the next digit of MAN#1 to the string buffer

Get the MSB of MAN#1

A040:lda &31

Get the higher nybble into the lower nybble

A042:lsr A**A043:lsr A****A044:lsr A****A045:lsr A**

Add it to the buffer as a digit

A046:jsr &A064

Remove the nybble from the mantissa

A049:lda &31**A04B:and #&F****A04D:sta &31**

Exit, multiplying MAN#1 by 10

A04F:jmp &A197**Output the exponent of a number**

This routine adds the accumulator to the string buffer as a positive number less than 100. This routine could be adapted to actually print the accumulator as a number between 0 to 99 by adding 'JMP &FFEE' at the end and substituting 'JSR &FFEE' for 'JSR &A064' at &A060

Set the most significant digit to -1

A052:ldx #&FF

Increment the digit and take 10 off the exponent

A054:sec**A055:inx****A056:sub #&A**

If no carry was generated, go back and repeat the process

A058:bcs &A055

Add the 10 back onto the accumulator to compensate for the final subtraction

A05A:adc #&A

Save the accumulator as the less significant digit

A05C:pha

Get the most significant digit into A

A05D:txa

Add it to the buffer as long as it is not zero

A05E:beq &A063

A060:jsr &A064

Retrieve the less significant digit

A063:pla

Make it into an ASCII digit

A064:ora #&30

Append a character to the string buffer

Save the X register

A066:stx &3B

Point to the first unused location of the buffer

A068:ldx &36

Store the character there

A06A:sta &600,X

Retrieve the X register

A06D:ldx &3B

Increment the string length counter

A06F:inc &36

Exit

A071:rts

Clear the carry flag to indicate failure

A072:clc

Set the rounding byte to zero

A073:stx &35

Set the sign and under/overflow byte to zero

A075:jsr &A1DA

Indicate the answer is a floating point number

A078:lda #&FF

Exit

A07A:rts

Get number from PTR #2

This routine translates an ASCII number at PTR #2 to a binary number either in the IAC or FAC #1. The type of the result is given in the accumulator on exit. Notice that only decimal numbers are translated

The routine uses two flags, EXPCNT and PNTFLG. EXPCNT is used to keep track of the current (decimal) exponent. PNTFLG is a flag which is set when a period has been encountered

The method used is to treat the number being built up as an integer, keeping track of the location of the decimal point. Then, to make it into a proper floating point number, it is divided or multiplied by 10 the required number of times

Zero MAN #1

A07B:ldx #&0

A07D:stx &31

A07F:stx &32

A081:stx &33

A083:stx &34

A085:stx &35

Zero EXPCNT

A087:stx &48

Zero PNTFLG

A089:stx &49

Jump forwards if the first character is a period

A08B:cmp #&2E

A08D:beq &A0A0

Exit if the first character is too big to be a digit

A08F:cmp #&3A

A091:bcs &A072

Take the ASCII factor off the digit

A093:sub #&2F

Exit if the character is too small to be a digit

A095:bmi &A072

Save the digit in the LSB of MAN#1

A097:sta &35

Get the next character

A099:iny

A09A:lda (&19),Y

Jump forwards if the character is not a period

A09C:cmp #&2E

A09E:bne &A0A8

Exit if a period has already been encountered

A0A0:lda &48

A0A2:bne &A0E8

Make PNTFLG be one to indicate the period has been found

A0A4:inc &48

Go back and examine the next character

A0A6:bne &A099

Jump forwards if the 'E' marking an exponent has been found

A0A8:cmp #&45

A0AA:beq &A0E1

Exit if the character is too big to be a digit

A0AC:cmp #&3A

A0AE:bcs &A0E8

Take off the ASCII factor

A0B0:sub #&2F

Exit if the character is too small to be a digit

A0B2:bcc &A0E8

Check if $MAN\#1$ is too big to accept another digit, by checking its MSB

A0B4:ldx &31

A0B6:cpx #&18

If it is still small enough, go to &A0C2

A0B8:bcc &A0C2

If a period has been encountered, simply go back for the next character

A0BA:ldx &48

A0BC:bne &A099

Otherwise, increment the exponent count

A0BE:inc &49

Go back and examine the next character

A0C0:bcx &A099

Decrement EXPCNT the period has been found

A0C2:ldx &48

A0C4:beq &A0C8

A0C6:dec &49

Set $MAN\#1 = MAN\#1 * 10$

A0C8:jsr &A197

Add in the new digit and go back for a new character

A0CB:adc &35

A0CD:sta &35

A0CF:bcc &A099

A0D1:inc &34

A0D3:bne &A099

A0D5:inc &33

A0D7:bne &A099

A0D9:inc &32

A0DB:bne &A099

A0DD:inc &31

A0DF:bne &A099

Get the exponent

A0E1:jsr &A140

Add it to the current exponent

A0E4:adc &49

A0E6:sta &49

Save the offset of PTR#2

A0E8:sty &1B

If the exponent is still given as zero, and no decimal point was found, exit, treating the number as an integer

A0EA:lda &49

A0EC:ora &48

A0EE:beq &A11F

Exit if the number is zero

A0F0:jsr &A1DA

A0F3:beq &A11B

Set the exponent to indicate that the bicimal point is to the right of the rounding byte

A0F5:lda #&A8

A0F7:sta &30

Zero the sign and over/underflow bytes

A0F9:lda #&0

A0FB:sta &2F

A0FD:sta &2E

Normalise the number

A0FF:jsr &A303

Get the current exponent

A102:lda &49

If it is negative, goto &A111

A104:bmi &A111

Exit if it is zero

A106:beq &A118

Multiply the number by 10

A108:jsr &A1F4

Decrement the exponent

A10B:dec &49

Continue adjusting if the exponent has not yet become zero

A10D:bne &A108

Otherwise, exit

A10F:beq &A118

Divide the number by 10

A111:jsr &A24D

Increment the exponent count

A114:inc &49

Continue this process until the exponent count becomes zero

A116:bne &A111

Test for overflow

A118:jsr &A65C

Exit, with the carry set to indicate success and A=&FF to indicate a floating point result

A11B:sec

A11C:lda #&FF

A11E:rts

Make sure the number is not too big to be an integer

A11F:lda &32

A121:sta &2D

A123:and #&80

A125:ora &31

A127:bne &A0F5

Transfer the other bytes from MAN#1 to the IAC

A129:lda &35**A12B:sta &2A****A12D:lda &34****A12F:sta &2B****A131:lda &33****A133:sta &2C**

Indicate the result is an integer

A135:lda #&40

Indicate that a number was found

A137:sec

Exit

A138:rts

Get the exponent

A139:jsr &A14B

Take the one's complement of it

A13C:eor #&FF

Set the carry, so that the ADC instruction on exit will turn it into a two's complement operation

A13E:sec

Exit

A13F:rts**Decode the 'E' part of a number**

Get the next character

A140:iny**A141:lda (&19),Y**

Run a different routine if the exponent starts with a '-'

A143:cmp #&2D**A145:beq &A139**

Get a new character if the current one is a '+', which are ignored

A147:cmp #&2B**A149:bne &A14E****A14B:iny****A14C:lda (&19),Y**

Exit with an exponent of zero if the character is too big to be a digit

A14E:cmp #&3A**A150:bcs &A174**

Take off the ASCII factor

A152:sub #&2F

Exit with zero if the character is too small to be a digit

A154:bcc &A174

Save the exponent

A156:sta &4A

Get the next character

A158:iny**A159:lda (&19),Y**

Exit with the first digit if the character is too big to be a digit

A15B:cmp #&3A**A15D:bcs &A170**

Take off the ASCII factor

A15F:sub #&2F

Exit with the first digit if the character is too small to be a digit

A161:bcc &A170

Increment past the second digit

A163:iny

Save the second digit

A164:sta &43

Multiply the first digit by 10

A166:lda &4A**A168:asl A****A169:asl A****A16A:adc &4A****A16C:asl A**

Add in the second digit

A16D:adc &43

Exit

A16F:rts

Exit with the first digit

A170:lda &4A**A172:clc****A173:rts**

Exit with zero

A174:lda #&0**A176:clc****A177:rts****Set MAN#1 = MAN#1 + MAN#2**

This routine adds MAN#1 to MAN#2, treating them as integers. Notice that, unlike normal integers, the least significant byte is stored higher up in memory

A178:lda &35**A17A:adc &42****A17C:sta &35****A17E:lda &34****A180:adc &41****A182:sta &34****A184:lda &33****A186:adc &40****A188:sta &33**


```

A18A:lda &32
A18C:adc &3F
A18E:sta &32

A190:lda &31
A192:adc &3E
A194:sta &31

A196:rts

```

Set MAN#1 = MAN#1*10

This routine multiplies MAN#1 by 10, treating it as an integer. It preserves the accumulator in the meantime

Save the accumulator

```
A197:pha
```

Save MAN#1 in X, A and the machine stack

```

A198:ldx &34
A19A:lda &31
A19C:pha
A19D:lda &32
A19F:pha
A1A0:lda &33
A1A2:pha
A1A3:lda &35

```

Multiply MAN#1 by two

```

A1A5:asl A
A1A6:rol &34
A1A8:rol &33
A1AA:rol &32
A1AC:rol &31

```

Multiply MAN#1 by two again, making four altogether

```

A1AE:asl A
A1AF:rol &34
A1B1:rol &33
A1B3:rol &32
A1B5:rol &31

```

Add the original value of MAN#1 to its new value, making a total multiplication of five

```

A1B7:adc &35
A1B9:sta &35
A1BB:txa
A1BC:adc &34
A1BE:sta &34
A1C0:pla
A1C1:adc &33
A1C3:sta &33

```

A1C5:pla**A1C6:adc &32****A1C8:sta &32****A1CA:pla****A1CB:adc &31**

Multiply MAN#1 by 2, making 10 altogether

A1CD:asl &35**A1CF:rol &34****A1D1:rol &33****A1D3:rol &32****A1D5:rol A****A1D6:sta &31**

Retrieve the accumulator

A1D8:pla

Exit

A1D9:rts**Set the flags for FAC#1**

This routine sets the sign and zero flags according to the contents of FAC#1

Check whether MAN#1 is zero

A1DA:lda &31**A1DC:ora &32****A1DE:ora &33****A1E0:ora &34****A1E2:ora &35**

Zero the sign, overflow and exponent if it is

A1E4:beq &A1ED

Otherwise, get the sign of the number

A1E6:lda &2E

Exit if it is not zero

A1E8:bne &A1F3

Otherwise, indicate that the result is positive and non-zero by getting one into the accumulator

A1EA:lda #&1

Exit

A1EC:rts

Zero the sign, exponent and under/overflow bytes

A1ED:sta &2E**A1EF:sta &30****A1F1:sta &2F**

Exit

A1F3:rts

Set FAC#1 = FAC#1*10

Add 3 to the exponent of FAC#1. This has the result of multiplying the number by 23, which is 8

A1F4:clc

A1F5:lda &30

A1F7:adc #&3

A1F9:sta &30

Increment the overflow byte if the exponent overflowed

A1FB:bcc &A1FF

A1FD:inc &2F

Copy FAC#1 to FAC#2

A1FF:jsr &A21E

Divide MAN#2 by four

A202:jsr &A242

A205:jsr &A242

Add MAN#2 to MAN#1. This has the result of setting $MAN\#1 = (MAN\#1 + MAN\#1/4)*8$, which simplifies to $MAN\#1*10$

A208:jsr &A178

If the addition did not overflow, exit

A20B:bcc &A21D

Shift MAN#1 right, setting the most significant bit

A20D:ror &31

A20F:ror &32

A211:ror &33

A213:ror &34

A215:ror &35

Increment the exponent to compensate

A217:inc &30

Increment the overflow byte if needed

A219:bne &A21D

A21B:inc &2F

Exit

A21D:rts

Set FAC#2 = FAC#1

A21E:lda &2E

A220:sta &3B

A222:lda &2F

A224:sta &3C

A226:lda &30

A228:sta &3D**A22A:lda &31****A22C:sta &3E****A22E:lda &32****A230:sta &3F****A232:lda &33****A234:sta &40****A236:lda &34****A238:sta &41****A23A:lda &35****A23C:sta &42****A23E:rts****Copy FAC#1 to FAC#2; set FAC#2 = FAC#2*2**

Copy FAC#1 to FAC#2

A23F:jsr &A21E

Shift MAN#2 right

A242:lsr &3E**A244:ror &3F****A246:ror &40****A248:ror &41****A24A:ror &42**

Exit

A24C:rts**Set FAC#1 = FAC#1/10**

This routine divides FAC#1 by 10 using floating point arithmetic

Subtract 4 from the exponent, so dividing FAC#1 by 16

A24D:sec**A24E:lda &30****A250:sbc #&4****A252:sta &30**

Adjust the underflow byte if needed

A254:bcs &A258**A256:dec &2F**

Copy MAN#1 to MAN#2 and divide MAN#2 by two

A258:jsr &A23F

Set MAN#1 = MAN#1 + MAN#2

A25B:jsr &A208

Copy MAN#1 to MAN#2 and divide MAN#2 by two

A25E:jsr &A23F

Divide MAN#2 by eight

```

A261:jsr &A242
A264:jsr &A242
A267:jsr &A242
Set MAN#1 = MAN#1 + MAN#2
A26A:jsr &A208
Set MAN#2 = MAN#1 DIV 256
A26D:lda #&0
A26F:sta &3E
A271:lda &31
A273:sta &3F
A275:lda &32
A277:sta &40
A279:lda &33
A27B:sta &41
A27D:lda &34
A27F:sta &42
Include the rounding bit of the rounding byte and set MAN #1 = MAN#1 + MAN
#2
A281:lda &35
A283:rol A
A284:jsr &A208
Set MAN#2 = MAN#1 DIV 65536
A287:lda #&0
A289:sta &3E
A28B:sta &3F
A28D:lda &31
A28F:sta &40
A291:lda &32
A293:sta &41
A295:lda &33
A297:sta &42
Set MAN#1 = MAN#1 + MAN#2, including the rounding bit from the mantissa
A299:lda &34
A29B:rol A
A29C:jsr &A208
Get the rounding bit of byte 2 of the mantissa into the carry flag
A29F:lda &32
A2A1:rol A
Get byte 1
A2A2:lda &31

Set MAN#1 = MAN#1 + A
This routine adds the contents of the accumulator to MAN#1
Add the least significant byte to the accumulator
A2A4:adc &35

```

A2A6:sta &35

Increment the next bit if a carry occurs

A2A8:bcc &A2BD**A2AA:inc &34****A2AC:bne &A2BD****A2AE:inc &33****A2B0:bne &A2BD****A2B2:inc &32****A2B4:bne &A2BD****A2B6:inc &31****A2B8:bne &A2BD**

Shift the number right if overflow occurs

A2BA:jmp &A20B**A2BD:rts****Convert the IAC to FAC#1**

This routine converts the integer in the IAC to a floating point number in FAC#1

Zero the under/overflow and rounding bytes of FAC#1

A2BE:ldx #&0**A2C0:stx &35****A2C2:stx &2F**

Skip negating the integer if it is already positive

A2C4:lda &2D**A2C6:bpl &A2CD**

Negate it

A2C8:jsr &AD93

Indicate the floating point sign

A2CB:ldx #&FF

Save the sign

A2CD:stx &2E

Move the body of the integer into the mantissa

A2CF:lda &2A**A2D1:sta &34****A2D3:lda &2B****A2D5:sta &33****A2D7:lda &2C****A2D9:sta &32****A2DB:lda &2D****A2DD:sta &31**

Set the exponent to &A0, which indicates that the bicemal point is 32 bits into the number, which is what you would expect for an integer

A2DF:lda #&A0**A2E1:sta &30**

Normalise the number

A2E3:jmp &A303

Zero rounding, sign and underflow bytes

This routine copies the accumulator into the sign, over/underflow and rounding bytes of FAC#1

```

A2E6:sta &2E
A2E8:sta &30
A2EA:sta &2F
A2EC:rts

```

Convert byte in A to FAC#1

This routine converts the byte in the accumulator into a floating point number in FAC#1

Save the accumulator

```
A2ED:pha
```

Zero FAC#1

```
A2EE:jsr &A686
```

Retrieve the accumulator

```
A2F1:pla
```

Exit if it is zero

```
A2F2:beq &A2EC
```

Skip to &A2FD if it is positive

```
A2F4:bpl &A2FD
```

Set the sign byte to indicate that FAC#1 is negative

```
A2F6:sta &2E
```

Negate the accumulator

```
A2F8:lda #&0
```

```
A2FA:sec
```

```
A2FB:sbc &2E
```

Save the accumulator as the most significant byte of the mantissa

```
A2FD:sta &31
```

Set the exponent to &88 which indicates that the bicimal point is to the right of the last bit of the byte

```
A2FF:lda #&88
```

```
A301:sta &30
```

Fall into the normalisation routine

Normalise FAC#1

Exit if the number is already normalised

```
A303:lda &31
```

```
A305:bmi &A2EC
```

If the rest of the mantissa is zero, zero the sign, under/overflow and rounding bytes and exit

```
A307:ora &32
```

```
A309:ora &33
```

```
A30B:ora &34
```

```
A30D:ora &35
```

A30F:beq &A2E6

Rescue the exponent

A311:lda &30

Exit if the number is normalised

A313:ldy &31**A315:bmi &A2EC**

If the least significant byte of the mantissa is not zero, no byte shifts are required to normalise the number, so go to &A33A to carry out the bit shifts

A317:bne &A33A

Carry out a byte shift on MAN#1

A319:ldx &32**A31B:stx &31****A31D:ldx &33****A31F:stx &32****A321:ldx &34****A323:stx &33****A325:ldx &35****A327:stx &34**

Zero the rounding byte

A329:sty &35

Subtract eight from the exponent

A32B:sec**A32C:sbc #&8****A32E:sta &30**

Return to the start of the loop, updating the underflow flag if necessary

A330:bcs &A313**A332:dec &2F****A334:bcc &A313**

Exit if the number is normalised

A336:ldy &31**A338:bmi &A2EC**

Shift the mantissa left by one bit

A33A:asl &35**A33C:rol &34****A33E:rol &33****A340:rol &32****A342:rol &31**

Decrement the accumulator (the ROL instruction at &A342 will always result in a clear carry flag)

A344:sbc #&0

Update the exponent

A346:sta &30

Return to the bit shift section, updating the underflow byte if necessary

A348:bcs &A336

A34A:dec &2F
A34C:bcc &A336

Unpack FAC#2 from (&4B)

This routine unpacks the five byte floating point number pointed to by (&4B) to FAC#2

Get the least significant byte of the mantissa of the number

A34E:ldy #&4
A350:lda (&4B),Y

Save it in FAC#2

A352:sta &41

Transfer the next byte

A354:dey
A355:lda (&4B),Y
A357:sta &40

Transfer the next byte

A359:dey
A35A:lda (&4B),Y
A35C:sta &3F

Transfer the least significant byte of the mantissa as the sign byte

A35E:dey
A35F:lda (&4B),Y
A361:sta &3B

Point to the exponent

A363:dey

Zero the over/underflow and rounding bytes

A364:sty &42
A366:sty &3C

Transfer the the exponent of the number

A368:lda (&4B),Y

If the exponent and the mantissa of the number are zero, then zero the least significant byte of the mantissa and exit

A36A:sta &3D
A36C:ora &3B
A36E:ora &3F
A370:ora &40
A372:ora &41
A374:beq &A37A

Get the least significant byte of the mantissa from the sign byte, setting the true numeric bit

A376:lda &3B
A378:ora #&80
A37A:sta &3E

Exit

A37C:rts

Pack FAC#1 to &471 onwards

Indicate that the LSB of the address is &71 and join the main routine

A37D:lda #&71

A37F:bne &A387

Pack FAC#1 to &476 onwards

Indicate that the LSB of the address is &76 and join the main routine

A381:lda #&76

A383:bne &A387

Pack FAC#1 to &46C onwards

Make (&4B) point to &46C

A385:lda #&6C

A387:sta &4B

A389:lda #&4

A38B:sta &4C

Pack FAC#1 to (&4B)

Transfer the exponent

A38D:ldy #&0

A38F:lda &30

A391:sta (&4B),Y

Point to the first byte of the mantissa

A393:iny

Get the sign byte from FAC#1

A394:lda &2E

Mask out all the bits except for the sign bit

A396:and #&80

A398:sta &2E

Get the first byte of the mantissa

A39A:lda &31

Replace the true numeric bit with the sign bit

A39C:and #&7F

A39E:ora &2E

Save the result

A3A0:sta (&4B),Y

Transfer the next byte

A3A2:lda &32

A3A4:iny

A3A5:sta (&4B),Y

Transfer the next byte

A3A7:lda &33

A3A9:iny

A3AA:sta (&4B),Y

Transfer the least significant byte

```

A3AC:lda &34
A3AE:iny
A3AF:sta (&4B),Y
A3B1:rts

```

Unpack FAC# 1 from &46C

Set (&4B) to &46C

```
A3B2:jsr &A7F5
```

Unpack FAC# 1 from (&4B)

Transfer the less significant three bytes of the mantissa

```

A3B5:ldy #&4
A3B7:lda (&4B),Y
A3B9:sta &34
A3BB:dey
A3BC:lda (&4B),Y
A3BE:sta &33
A3C0:dey
A3C1:lda (&4B),Y
A3C3:sta &32

```

Transfer the most significant byte of the mantissa to the sign byte

```

A3C5:dey
A3C6:lda (&4B),Y
A3C8:sta &2E

```

Transfer the exponent

```

A3CA:dey
A3CB:lda (&4B),Y
A3CD:sta &30

```

Zero the under/overflow and rounding bytes

```

A3CF:sty &35
A3D1:sty &2F

```

Zero the top byte of the mantissa if it is zero

```

A3D3:ora &2E
A3D5:ora &32
A3D7:ora &33
A3D9:ora &34
A3DB:beq &A3E1

```

Replace the sign bit with a true numeric bit and place it in the mantissa

```

A3DD:lda &2E
A3DF:ora #&80
A3E1:sta &31

```

Exit

```
A3E3:rts
```

Convert FAC#1 to the IAC

This routine converts the floating point number in FAC#1 to an integer in the IAC

Convert the floating point number to an integer

A3E4:jsr &A3FE

Transfer the result to the IAC

A3E7:lda &31

A3E9:sta &2D

A3EB:lda &32

A3ED:sta &2C

A3EF:lda &33

A3F1:sta &2B

A3F3:lda &34

A3F5:sta &2A

Exit

A3F7:rts

Copy FAC#1 to FAC#2

A3F8:jsr &A21E

Set FAC#1 to zero and exit

A3FB:jmp &A686

Fix FAC#1

This routine converts FAC#1 to an integer, leaving the result in MAN#1. The fractional part of the number is left in MAN#2.

If the exponent is less than &80, the number is less than 1. Thus, the integer part is zero and the fractional part is FAC#1

A3FE:lda &30

A400:bpl &A3F8

Zero FAC#2

A402:jsr &A453

Check FAC#1 for zero

A405:jsr &A1DA

Fix it if it is not zero

A408:bne &A43C

Exit if it is zero

A40A:beq &A468

If the exponent is greater than or equal to &A0, the number is now an integer, or is too big to ever be an integer, in which case the routine must exit

A40C:lda &30

A40E:cmp #&A0

A410:bcs &A466

If the exponent is greater than &99, only a series of bit shifts are required, so branch to &A43C to carry these out

A412:cmp #&99

A414: bcs &A43C

Add eight to the exponent to allow for the shift by one byte

A416: adc #&8

A418: sta &30

Shift MAN#1 and MAN#2 right by one byte

A41A: lda &40

A41C: sta &41

A41E: lda &3F

A420: sta &40

A422: lda &3E

A424: sta &3F

A426: lda &34

A428: sta &3E

A42A: lda &33

A42C: sta &34

A42E: lda &32

A430: sta &33

A432: lda &31

A434: sta &32

Zero the least significant byte of MAN#1

A436: lda #&0

A438: sta &31

Return to repeat the tests

A43A: beq &A40C

Shift MAN#1 and MAN#2 right one bit

A43C: lsr &31

A43E: ror &32

A440: ror &33

A442: ror &34

A444: ror &3E

A446: ror &3F

A448: ror &40

A44A: ror &41

Increment the exponent to compensate

A44C: inc &30

Continue and repeat the test

A44E: bne &A40C

Direct jump to the 'Too big' error message

A450: jmp &A66C

Zero FAC#2

A453: lda #&0

A455: sta &3B

A457: sta &3C

A459: sta &3D

A45B:sta &3E**A45D:sta &3F****A45F:sta &40****A461:sta &41****A463:sta &42****A465:rts**

If the exponent is not actually equal to &A0, give a 'Too big' error message

A466:bne &A450**Negate MAN#1 if necessary**

This routine negates MAN#1 if it is negative

Exit if the number is positive

A468:lda &2E**A46A:bpl &A485**

Negate MAN#1 by subtracting it from zero

A46C:sec**A46D:lda #&0****A46F:sbc &34****A471:sta &34****A473:lda #&0****A475:sbc &33****A477:sta &33****A479:lda #&0****A47B:sbc &32****A47D:sta &32****A47F:lda #&0****A481:sbc &31****A483:sta &31**

Exit

A485:rts**Return the fractional part of FAC#1**

This routine returns the fractional part of FAC#1 in FAC#1 and the integral part (between -128 and 127) in &4A. The integral part is rounded to the nearest integer. The fractional part is signed in such a way that if added to the integral part, the original number would result. Thus, the fractional part will lie between -0.5 and 0.5.

If the exponent is greater than or equal to &80, the fixing operation is required since the number is greater than 1. Otherwise, FAC#1 is already the fractional part of the number

A486:lda &30**A488:bmi &A491**

Zero the integral part of the number

A48A:lda #&0**A48C:sta &4A**

Exit, checking FAC#1 for zero

A48E: jmp &A1DA

Fix FAC#1

A491: jsr &A3FE

Get the least significant integer part

A494: lda &34

A496: sta &4A

Copy MAN#2 to MAN#1

A498: jsr &A4E8

Set the exponent to &80, which puts the bicemal point at the start of the number

A49B: lda #&80

A49D: sta &30

If the fractional part of the number is less than 0.5, normalise and exit. In this case, no rounding is required

A49F: ldx &31

A4A1: bpl &A4B3

Invert the sign of the fractional part

A4A3: eor &2E

A4A5: sta &2E

If the fractional part is now positive, the integral part must be rounded downwards, since it is negative

A4A7: bpl &A4AE

Otherwise, it is rounded upwards

A4A9: inc &4A

Skip the next instruction

A4AB: jmp &A4B0

Round the integral part downwards

A4AE: dec &4A

Negate MAN#1

A4B0: jsr &A46C

Normalise and exit

A4B3: jmp &A303

Increment MAN#1

A4B6: inc &34

A4B8: bne &A4C6

A4BA: inc &33

A4BC: bne &A4C6

A4BE: inc &32

A4C0: bne &A4C6

A4C2: inc &31

Give a 'Too big' error message if the result overflows

A4C4: beq &A450

Exit

A4C6: rts

Decrement MAN#1

Negate MAN#1

A4C7:jsr &A46C

Increment MAN#1

A4CA:jsr &A4B6

Negate MAN#1 and exit

A4CD:jmp &A46C

Set FAC#1 = FAC#1-(&4B)

Set FAC#1 = (&4B)-FAC#1

A4D0:jsr &A4FD

Negate the result and exit

A4D3:jmp &AD7E

Exchange FAC#1 and (&4B)

Get FAC#2 from (&4B)

A4D6:jsr &A34E

Copy FAC#1 to (&4B)

A4D9:jsr &A38D

Copy FAC#2 to FAC#1

Set FAC#1 = FAC#2

A4DC:lda &3B

A4DE:sta &2E

A4E0:lda &3C

A4E2:sta &2F

A4E4:lda &3D

A4E6:sta &30

Set MAN#1 = MAN#2

A4E8:lda &3E

A4EA:sta &31

A4EC:lda &3F

A4EE:sta &32

A4F0:lda &40

A4F2:sta &33

A4F4:lda &41

A4F6:sta &34

A4F8:lda &42

A4FA:sta &35

A4FC:rts

Set FAC#1 = (&4B)-FAC#1

Negate FAC#1

A4FD:jsr &AD7E

Fall into addition routine

Set FAC#1 = (&4B) + FAC#1

Get FAC#2 from (&4B)

A500:jsr &A34E

If FAC#2 is zero, the answer is already in FAC#1, so exit

A503:beq &A4FC

Do the addition

A505:jsr &A50B

Exit, checking the over/underflow byte

A508:jmp &A65C

Set FAC#1 = FAC#1 + FAC#2

Refer to the explanation of floating point arithmetic while reading this routine

Check whether FAC#1 is zero

A50B:jsr &A1DA

If FAC#1 is zero, the result is in FAC#2, so the routine must exit and copy FAC#1

A50E:beq &A4DC

Initialise Y with zero. This allows memory locations to be zeroed with an 'STY' instruction

A510:ldy #&0

Subtract EXP#2 from EXP#1

A512:sec

A513:lda &30

A515:sub &3D

If the result is zero, no shifts are needed, so goto &A590

A517:beq &A590

If EXP#1 < EXP#2 then goto &A552 and shift FAC#1

A519:bcc &A552

If the exponent difference is over &25, FAC#2 is so small (compared to FAC#1) that FAC#1 + FAC#2 is equal to FAC#1. In this case the routine can exit now, since the result is in FAC#1

A51B:cmp #&25

A51D:bcs &A4FC

Save the exponent difference

A51F:pha

Use the AND operation to find out the number of bytes that FAC#2 should be shifted by

A520:and #&38

If no bytes need to be moved, skip to the bit shift section at &A53D

A522:beq &A53D

Divide the number of bytes to be shifted by eight. This removes the three insignificant bits giving the number of bit shifts

A524:lsr A

A525:lsr A

A526:lsr A

Transfer the result to X to use as a counter

A527:tax

Shift MAN#2 one byte to the right

A528:lda &41**A52A:sta &42****A52C:lda &40****A52E:sta &41****A530:lda &3F****A532:sta &40****A534:lda &3E****A536:sta &3F****A538:sty &3E**

Continue the process for the indicated number of shifts

A53A:dex**A53B:bne &A528**

Retrieve the exponent difference

A53D:pla

Use the AND operation to extract the number of bit shifts required

A53E:and #&7

The numbers are properly lined up if the result is zero

A540:beq &A590

Get the number into X to be used as a counter

A542:tax

Shift MAN#2 right one bit

A543:lsr &3E**A545:ror &3F****A547:ror &40****A549:ror &41****A54B:ror &42**

Continue the process for the indicated number of bits

A54D:dex**A54E:bne &A543**

Skip to &A590

A550:beq &A590

Subtract EXP#1 from EXP#2

A552:sec**A553:lda &3D****A555:sbc &30**

If the result is over &25, exit and use FAC#2 as the result

A557:cmp #&25**A559:bcs &A4DC**

Save the exponent difference

A55B:pha

Extract the number of byte shifts required

A55C:and #&38

Skip to the bit shift section if the result is zero

A55E:beq &A579

Line the number up correctly

A560:lsr A

A561:lsr A

A562:lsr A

Transfer the number to X so that it can be used as a counter

A563:tax

Carry out a byte shift on MAN#1

A564:lda &34

A566:sta &35

A568:lda &33

A56A:sta &34

A56C:lda &32

A56E:sta &33

A570:lda &31

A572:sta &32

A574:sty &31

Continue for the required number of shifts

A576:dex

A577:bne &A564

Retrieve the exponent difference

A579:pla

Extract the number of bit shifts required

A57A:and #&7

Exit if the result is zero

A57C:beq &A58C

A57E:tax

Carry out the bit shift

A57F:lsr &31

A581:ror &32

A583:ror &33

A585:ror &34

A587:ror &35

Continue for the required number

A589:dex

A58A:bne &A57F

Set EXP#1 to EXP#2 to show the exponent has been adjusted

A58C:lda &3D

A58E:sta &30

Exclusive-OR the signs of the two numbers

A590:lda &2E

A592:eor &3B

If the result is positive, the two numbers must have the same sign. This means they can be added directly

A594:bpl &A5DF

Compare the lower bytes of the mantissas, skipping to &A5B7 if they are not equal

A596:lda &31

A598:cmp &3E

A59A:bne &A5B7

Compare the next bytes

A59C:lda &32

A59E:cmp &3F

A5A0:bne &A5B7

Compare the next bytes

A5A2:lda &33

A5A4:cmp &40

A5A6:bne &A5B7

Compare the next bytes

A5A8:lda &34

A5AA:cmp &41

A5AC:bne &A5B7

Compare the next bytes

A5AE:lda &35

A5B0:cmp &42

A5B2:bne &A5B7

If the two mantissas are the same magnitude, but of a different sign, the result is zero

A5B4:jmp &A686

Goto &A5E3 if MAN#1 > MAN#2

A5B7:bcs &A5E3

Set MAN#1 = MAN#2 - MAN#1

A5B9:sec

A5BA:lda &42

A5BC:sbc &35

A5BE:sta &35

A5C0:lda &41

A5C2:sbc &34

A5C4:sta &34

A5C6:lda &40

A5C8:sbc &33

A5CA:sta &33

A5CC:lda &3F

A5CE:sbc &32

A5D0:sta &32

A5D2:lda &3E

A5D4:sbc &31

A5D6:sta &31

Use the sign of FAC#2 as the sign of the result

A5D8:lda &3B

A5DA:sta &2E

Normalise the result and exit

A5DC: jmp &A303

If the two numbers are of the same sign, add them and exit

A5DF: clc

A5E0: jmp &A208

Set $MAN\#1 = MAN\#1 - MAN\#2$

A5E3: sec

A5E4: lda &35

A5E6: sbc &42

A5E8: sta &35

A5EA: lda &34

A5EC: sbc &41

A5EE: sta &34

A5F0: lda &33

A5F2: sbc &40

A5F4: sta &33

A5F6: lda &32

A5F8: sbc &3F

A5FA: sta &32

A5FC: lda &31

A5FE: sbc &3E

A600: sta &31

Normalise and exit

A602: jmp &A303

Exit

A605: rts

Set $FAC\#1 = FAC\#1 \star (&4B)$

Check whether $FAC\#1$ is zero

A606: jsr &A1DA

Exit if it is, since this means the result should be zero

A609: beq &A605

Unpack $(&4B)$ to $FAC\#2$

A60B: jsr &A34E

Exit with the result as zero if $FAC\#2$ is zero

A60E: bne &A613

A610: jmp &A686

Set $FAC\#1 = FAC\#1 \star FAC\#2$

Add the exponents of the two numbers

A613: clc

A614: lda &30

A616: adc &3D

Increment the overflow flag if overflow occurred when adding the exponents

A618: bcc &A61D

A61A:inc &2F

Take &80 off the sum of the exponents

A61C:clc

A61D:sub #&7F

Save the result as the exponent of the result

A61F:sta &30

Indicate underflow if necessary

A621:bcs &A625

A623:dec &2F

Copy MAN#1 to &42-&47 (which will be referred to as MAN#3) and zero MAN#1

A625:ldx #&5

A627:ldy #&0

A629:lda &30,X

A62B:sta &42,X

A62D:sty &30,X

A62F:dex

A630:bne &A629

Exclusive-OR the signs of the two numbers and use the result as the sign of the result

A632:lda &2E

A634:eor &3B

A636:sta &2E

Set the loop counter

A638:ldy #&20

Shift MAN#2 right one bit

A63A:lsr &3E

A63C:ror &3F

A63E:ror &40

A640:ror &41

A642:ror &42

Shift MAN#2 left one bit

A644:asl &46

A646:rol &45

A648:rol &44

A64A:rol &43

Skip the addition if a carry did not 'fall out' of MAN#2

A64C:bcc &A652

Add MAN#2 to MAN#1

A64E:clc

A64F:jsr &A178

Continue the process

A652:dey

A653:bne &A63A

Exit

A655:rts

Set FAC#1 = FAC#1 ★ (&4B) & check overflow

Do the multiplication

A656:jsr &A606

Normalise the result

A659:jsr &A303

If the rounding byte is less than &80, then goto &A67C

A65C:lda &35

A65E:cmp #&80

A660:bcc &A67C

If it is equal to &80, then goto &A676

A662:beq &A676

Thus, the rounding byte is over &80 when control passes here. Add &FF to the mantissa, to round the result

A664:lda #&FF

A666:jsr &A2A4

Skip the error message

A669:jmp &A67C

'Too big' error message

A66C:brk

A66D:14-

A66E:54-T

A66F:6F-o

A670:6F-o

A671:20-

A672:62-b

A673:69-i

A674:67-g

A675:00-

Set the least significant bit of the mantissa, as a partial rounding operation

A676:lda &34

A678:ora #&1

A67A:sta &34

Zero the rounding byte

A67C:lda #&0

A67E:sta &35

Exit if the over/underflow byte is zero

A680:lda &2F

A682:beq &A698

Give a 'Too big' error message if it is positive

A684:bpl &A66C

Otherwise, zero FAC#1

Set FAC#1 = 0

A686:lda #&0

A688:sta &2E

A68A:sta &2F**A68C:sta &30****Set MAN#1 = A****A68E:sta &31****A690:sta &32****A692:sta &33****A694:sta &34****A696:sta &35****A698:rts****Set FAC#1 = 1**

Zero FAC#1

A699:jsr &A686

Set the mantissa to &80000000

A69C:ldy #&80**A69E:sty &31**

Set the exponent to &81

A6A0:iny**A6A1:sty &30**

Exit

A6A3:tya**A6A4:rts****Set FAC#1 = 1/FAC#1**

Save FAC#1 at &46C onwards

A6A5:jsr &A385

Set FAC#1 = 1

A6A8:jsr &A699

Set FAC#1 = 1/(&4B) and exit

A6AB:bne &A6E7**Set FAC#1 = (&4B)/FAC#1**

Give a 'Divide by zero' error message if FAC#1 is zero

A6AD:jsr &A1DA**A6B0:beq &A6BB**

Copy FAC#1 to FAC#2

A6B2:jsr &A21E

Unpack FAC#1 from (&4B)

A6B5:jsr &A3B5

Set FAC#1 = FAC#1/FAC#2 and exit if FAC#1 is not zero

A6B8:bne &A6F1

Exit with zero as the result

A6BA:rts

Direct jump to the 'Division by zero' error message

A6BB: jmp &99A7

TAN function routine

The TAN function is computed using the formula $\text{TAN}(X) = \text{SIN}(X) / \text{COS}(X)$

Get the operand and ensure it is a real number

A6BE: jsr &92FA

Compute the quadrant of the angle

A6C1: jsr &A9D3

Save the quadrant on the stack

A6C4: lda &4A

A6C6: pha

Set (&4B) to &47B

A6C7: jsr &A7E9

Save FAC #1 at (&4B)

A6CA: jsr &A38D

Increment the quadrant to allow the cosine to be taken

A6CD: inc &4A

Call the COS routine

A6CF: jsr &A99E

Set (&4B) to &47B

A6D2: jsr &A7E9

Swap FAC #1 with &47B

A6D5: jsr &A4D6

Retrieve the quadrant

A6D8: pla

A6D9: sta &4A

Call the SIN routine

A6DB: jsr &A99E

Set (&4B) point to &47B

A6DE: jsr &A7E9

Set $\text{FAC} \#1 = \text{FAC} \#1 / (\&4B)$

A6E1: jsr &A6E7

Indicate that the result is a real number

A6E4: lda #&FF

Exit

A6E6: rts

Set $\text{FAC} \#1 = \text{FAC} \#1 / (\&4B)$

Check if $\text{FAC} \#1$ is zero

A6E7: jsr &A1DA

Exit if it is, leaving the result as zero

A6EA: beq &A698

Unpack $\text{FAC} \#2$ from (&4B)

A6EC: jsr &A34E

Give a 'Divide by zero' error message if $\text{FAC} \#2$ is zero

A6EF:beq &A6BB

Fall into the main division code

Set FAC#1 = FAC#1/FAC#2

Exclusive-OR the signs of the two numbers, to get the sign of the result

A6F1:lda &2E

A6F3:eor &3B

A6F5:sta &2E

Take EXP#2 away from EXP#1

A6F7:sec

A6F8:lda &30

A6FA:sbc &3D

Adjust the underflow flag if necessary

A6FC:bcs &A701

A6FE:dec &2F

Add &81 to the result and use it as the exponent of the answer

A700:sec

A701:adc #&80

A703:sta &30

Adjust the overflow flag if necessary

A705:bcc &A70A

A707:inc &2F

Make sure the carry flag is clear to prevent the first BCS instruction being executed

A709:clc

Load the loop counter

A70A:ldx #&20

Skip the test if the previous shift set the carry flag

A70C:bcs &A726

Compare MAN#1 with MAN#2

A70E:lda &31

A710:cmp &3E

A712:bne &A724

A714:lda &32

A716:cmp &3F

A718:bne &A724

A71A:lda &33

A71C:cmp &40

A71E:bne &A724

A720:lda &34

A722:cmp &41

Skip the subtraction if MAN#1 < MAN#2

A724:bcc &A73F

Set $\text{MAN}\#1 = \text{MAN}\#1 - \text{MAN}\#2$

A726:lda &34

A728:sbc &41

A72A:sta &34

A72C:lda &33

A72E:sbc &40

A730:sta &33

A732:lda &32

A734:sbc &3F

A736:sta &32

A738:lda &31

A73A:sbc &3E

A73C:sta &31

Set the carry to indicate that a subtraction took place

A73E:sec

Shift a zero into ' $\text{MAN}\#3$ ' if the subtraction did not take place (the branch from &A724 will arrive here with the carry flag clear), or shift a one in if the subtraction took place

A73F:rol &46

A741:rol &45

A743:rol &44

A745:rol &43

Multiply $\text{MAN}\#1$ by two

A747:asl &34

A749:rol &33

A74B:rol &32

A74D:rol &31

Repeat the process as often as necessary

A74F:dex

A750:bne &A70C

Here, the same process is repeated except the rounding byte is used

Indicate that seven iterations are required

A752:ldx #&7

Skip the comparison if the previous shift generated a carry

A754:bcs &A76E

Compare $\text{MAN}\#2$ with $\text{MAN}\#1$

A756:lda &31

A758:cmp &3E

A75A:bne &A76C

A75C:lda &32
A75E:cmp &3F
A760:bne &A76C

A762:lda &33
A764:cmp &40
A766:bne &A76C

A768:lda &34
A76A:cmp &41

Skip the subtraction if $MAN\#1 < MAN\#2$

A76C:bcc &A787

Set $MAN\#1 = MAN\#1 - MAN\#2$

A76E:lda &34
A770:sbc &41
A772:sta &34

A774:lda &33
A776:sbc &40
A778:sta &33

A77A:lda &32
A77C:sbc &3F
A77E:sta &32

A780:lda &31
A782:sbc &3E
A784:sta &31

Indicate that the subtraction took place

A786:sec

Shift the carry flag (which indicates whether a subtraction took place) into the rounding byte of the result

A787:rol &35

Multiply the rest of $MAN\#1$ by two

A789:asl &34
A78B:rol &33
A78D:rol &32
A78F:rol &31

Continue for as many times as necessary

A791:dex
A792:bne &A754

Since only seven iterations were performed, this instruction is needed to line the rounding byte up

A794:asl &35

Transfer the result from 'MAN#3' to $MAN\#1$

A796:lda &46
A798:sta &34
A79A:lda &45
A79C:sta &33
A79E:lda &44
A7A0:sta &32
A7A2:lda &43
A7A4:sta &31

Check the over/underflow flag and exit

A7A6:jmp &A659

Give a '-ve root' error message

A7A9:brk
A7AA:15-
A7AB:2D--
A7AC:76-v
A7AD:65-e
A7AE:20-
A7AF:72-r
A7B0:6F-o
A7B1:6F-o
A7B2:74-t
A7B3:00-

SQR function routine

The square root routine uses the Newton method. For each iteration, the number of significant figures in the answer is doubled, allowing the computer to only perform 5 iterations, which gives fast results. Notice that many other versions of BASIC compute SQR as X0.5, which is considerably slower

The formula is:

$$X[i+1] = 0.5 * (X[i] + N/X[i])$$

In this formula, 'N' is the number whose root is being sought, while Xi is the previous root value and Xi+1 is the next value. Once the formula is converted to assembly language, it becomes rather more opaque

Evaluate the operand

A7B4:jsr &92FA

Check if it is zero

A7B7:jsr &A1DA

Exit with the result as zero if the argument is zero

A7BA:beq &A7E6

Give a '-ve root' error message if the argument is negative

A7BC:bmi &A7A9

Pack FAC #1 to &46C

A7BE:jsr &A385

Halve the exponent of the number by shifting it right and adding &40. This slightly complex procedure has to be adopted because of the bias given to the exponent. The new number is the first approximation to the value of the root

A7C1:lda &30**A7C3:lsr A****A7C4:adc #&40****A7C6:sta &30**

Set the loop counter

A7C8:lda #&5**A7CA:sta &4A**

Pack FAC#1 (which is the previous root value) to &471

A7CC:jsr &A7ED**A7CF:jsr &A38D**

Make (&4B) point to the argument of the function

A7D2:lda #&6C**A7D4:sta &4B**

Set FAC#1 = &46C/FAC#1. This corresponds to the 'N/X[i]' part of the formula above

A7D6:jsr &A6AD

Point to 'X[i]' with (&4B)

A7D9:lda #&71**A7DB:sta &4B**

Set FAC#1 = (&4B) + FAC#1

A7DD:jsr &A500

Decrement the exponent to divide the number by two

A7E0:dec &30

Continue for the required number of iterations

A7E2:dec &4A**A7E4:bne &A7CF**

Indicate that the result is real

A7E6:lda #&FF

Exit

A7E8:rts

Set (&4B) = &47B

A7E9:lda #&7B**A7EB:bne &A7F7**

Set (&4B) = &471

A7ED:lda #&71**A7EF:bne &A7F7**

Set (&4B) = &476

A7F1:lda #&76
 A7F3:bne &A7F7

Set (&4B) = &46C

A7F5:lda #&6C
 A7F7:sta &4B
 A7F9:lda #&4
 A7FB:sta &4C
 A7FD:rts

LN function routine

Evaluate the argument and ensure it is real

A7FE:jsr &92FA

Give a 'Log range' error message if the argument is negative or zero

A801:jsr &A1DA
 A804:beq &A808
 A806:bpl &A814
 A808:brk
 A809:16-
 A80A:4C-L
 A80B:6F-o
 A80C:67-g
 A80D:20-
 A80E:72-r
 A80F:61-a
 A810:6E-n
 A811:67-g
 A812:65-e
 A813:00-

Zero FAC#2

A814:jsr &A453

Set FAC#2 to one (notice that Y contains the exponent of FAC#2 after these instructions)

A817:ldy #&80
 A819:sty &3B
 A81B:sty &3E
 A81D:iny
 A81E:sty &3D

If the exponent of the argument is zero, skip the next test (notice that X contains the exponent of FAC#1 after these instructions)

A820:ldx &30
 A822:beq &A82A

Skip to &A82C if MAN#1 is less than &B5000000

A824:lda &31
 A826:cmp #&B5

A828:bcc &A82C

Increment X and Y

A82A:inx

A82B:dey

Save X on the machine stack

A82C:txa

A82D:pha

Save Y as the exponent of the result

A82E:sty &30

Set $FAC\#1 = FAC\#1 + FAC\#2$

A830:jsr &A505

Copy $FAC\#1$ to &47B

A833:lda #&7B

A835:jsr &A387

Load the address of the LN series table in A(lsb) and Y(msb)

A838:lda #&73

A83A:ldy #&A8

Call the series evaluator

A83C:jsr &A897

Make (&4B) point to &47B

A83F:jsr &A7E9

Set $FAC\#1 = FAC\#1 * (&4B) * (&4B)$

A842:jsr &A656

A845:jsr &A656

Set $FAC\#1 = FAC\#1 + (&4B)$

A848:jsr &A500

Pack $FAC\#1$ to &46C

A84B:jsr &A385

Retrieve the exponent of the argument

A84E:pla

Take off the bias

A84F:sec

A850:sbc #&81

Float the exponent into $FAC\#1$

A852:jsr &A2ED

Set $FAC\#1 = FAC\#1 * 0.693147181$

A855:lda #&6E

A857:sta &4B

A859:lda #&A8

A85B:sta &4C

A85D:jsr &A656

Make (&4B) point to &46C

A860:jsr &A7F5

Set $FAC\#1 = FAC\#1 + (&4B)$

A863:jsr &A500

Indicate that the result is real

A866:lda #&FF

Exit

A868:rts

Constants

This is one area of memory where numerical constants and values for the series evaluator are stored. To find the values of the constants a program like this was used:

> LIST

```

10 REM Filename:G-REAL
20
30 REM Examine BASIC real numbers
40
50 REM (c) 1983 Jeremy Ruston
60
70 A = 0
80 INPUT "Enter address:" A$
90 D% = EVAL(A$)
100 FOR T% = 0 TO 4
110 ?(TOP + 3 + T%) = ?(D% + T%)
120 NEXT T%
130 PRINT A

```

>

The program relies on the fact that the value of 'A' is stored at TOP + 3.

This constant, 0.434294482, is used to convert natural logarithms to base 10 logarithms. Refer to the 'LOG' function routine at &ABA8 for more details

A869:7F-

A86A:5E-^

A86B:5B-[

A86C:D8-X

A86D:AA-*

This constant, 0.693147181, is used in the computation of natural logarithms. If this number is 'x', then e^x is two, where 'e' is 2.718281828

A86E:80-

A86F:31-1

A870:72-r

A871:17-

A872:F8-x

The LN series

This series of seven constants are used in the series evaluator for LN.

This byte indicates that there are seven constants in the series

A873:06-

The first constant is 8.92463796E-3

A874:7A-z

A875:12-

A876:38-8

A877:A5-%

A878:0B-

The second constant is 249.057128

A879:88-

A87A:79-y

A87B:0E-

A87C:9F-

A87D:F3-s

The third constant is 4.16681756E-2

A87E:7C-|

A87F:2A-*

A880:AC-,

A881:3F-?

A882:B5-5

The fourth constant is 45.0015964

A883:86-

A884:34-4

A885:01-

A886:A2-"

A887:7A-z

The fifth constant is 0.444444416

A888:7F-

A889:63-c

A88A:8E-

A88B:37-7

A88C:EC-1

The sixth constant is 2.99999994

A88D:82-

A88E:3F-?

A88F:FF-

A890:FF-

A891:C1-A

The final constant is -0.4999999999

A892:7F-

A893:FF-

A894:FF-

A895:FF-

A896:FF-

Series evaluator

This routine is used by most of the transcendental functions to compute the series required. On entry, A(lsb) and Y(msb) must point to a series table. The table should contain a number (n) followed by n + 1 five byte constants.

This is a BASIC version of the series evaluator:

> LIST

```

10 REM Filename: SERDEMO
20
30 REM BASIC version of series evaluator
40
50 REM (c) 1983 Jeremy Ruston & Acorn
60
70 S = 0
80 INPUT "Enter address:" A$
90 D% = EVAL(A$)
100 INPUT "Enter starting value:" FAC1
110
120 S46C = FAC1
130 I% = ?D%:D% = D% + 1
140 FAC1 = FNnext
150 FAC1 = S46C/FAC1
160 D% = D% + 5
170 FAC1 = FAC1 + FNnext
180 I% = I%-1
190 IF I% < > 0 THEN GOTO 150
200 PRINT FAC1
210 END
220
230 DEF FNnext
240 LOCAL T%
250 FOR T% = 0 TO 3
260 ?(TOP + 3 + T%) = D%?T%
270 NEXT T%
280 = S

```

>

In fact, this can be simplified to:

$$A/(A/(A/N1 + N2) + N3) + N4$$

(The above formula shows a case where there are 4 constants in the series. 'A' represents the contents of FAC#1 on entry)

Save the address of the series table at (&4D)

A897:sta &4D

A899:sty &4E

Save FAC #1 at &46C

A89B:jsr &A385

Get the iteration count

A89E:ldy #&0

A8A0:lda (&4D),Y

A8A2:sta &48

Increment (&4D) past the iteration count

A8A4:inc &4D

A8A6:bne &A8AA

A8A8:inc &4E

Copy (&4D) to (&4B). (&4B) now points to the first constant in the table

A8AA:lda &4D

A8AC:sta &4B

A8AE:lda &4E

A8B0:sta &4C

Unpack the constant at (&4B) to FAC #1

A8B2:jsr &A3B5

Make (&4B) point to &46C

A8B5:jsr &A7F5

Set FAC #1 = (&4B)/FAC #1

A8B8:jsr &A6AD

Add five to (&4D), to point to the next constant. The result is also copied in (&4B)

A8BB:clc

A8BC:lda &4D

A8BE:adc #&5

A8C0:sta &4D

A8C2:sta &4B

A8C4:lda &4E

A8C6:adc #&0

A8C8:sta &4E

A8CA:sta &4C

Set FAC #1 = FAC #1 + (&4B)

A8CC:jsr &A500

Continue until the loop terminates

A8CF:dec &48

A8D1:bne &A8B5

Exit

A8D3:rts

ACS function routine

Execute the ASN code

A8D4:jsr &A8DA

Join the code at the end of ATN to turn the ASN into an ACS (ACS(X) = -ASN(X) + 1.5708 approximately)

A8D7:jmp &A927

ASN function routine

ASN is given by $ASN(x) = ATN(x/SQR(-x*x + 1))$

Evaluate the operand

A8DA:jsr &92FA

Skip to &A8EA if it is positive

A8DD:jsr &A1DA

A8E0:bpl &A8EA

Make the number positive by ensuring the sign bit is not set

A8E2:lsr &2E

Call the positive ASN code

A8E4:jsr &A8EA

Make the number negative and exit

A8E7:jmp &A916

Pack FAC # 1 to &476 onwards

A8EA:jsr &A381

Call the SIN routine to evaluate $SQR(x*x-1)$

A8ED:jsr &A9B1

Check the result

A8F0:jsr &A1DA

Skip if the result is zero

A8F3:beq &A8FE

Make (&4B) point to &476

A8F5:jsr &A7F1

Set $FAC \# 1 = (&4B)/FAC \# 1$

A8F8:jsr &A6AD

Take the ATN of the result

A8FB:jmp &A90A

Make (&4B) point to PI/2

A8FE:jsr &AA55

Get PI/2 into $FAC \# 1$

A901:jsr &A3B5

Indicate that the result is real

A904:lda #&FF

Exit

A906:rts

ATN function routine

Evaluate the operand

A907:jsr &92FA

Return if the argument is zero

A90A:jsr &A1DA

A90D:beq &A904

Skip forwards if the argument is positive

A90F:bpl &A91B

Make the argument positive by resetting the sign bit

A911:lsr &2E

Call the main ATN code

A913:jsr &A91B

Make the number positive

A916:lda #&80

A918:sta &2E

Exit

A91A:rts

Jump forwards if the argument is less than one

A91B:lda &30

A91D:cmp #&81

A91F:bcc &A936

Set $FAC\#1 = 1/FAC\#1$

A921:jsr &A6A5

Call the code for the argument being less than one

A924:jsr &A936

Set $FAC\#1 = FAC\#1 - 1.57080078$

A927:jsr &AA48

A92A:jsr &A500

Set $FAC\#1 = FAC\#1 + 4.45445511E-6$

A92D:jsr &AA4C

A930:jsr &A500

Exit, negating the result

A933:jmp &AD7E

Exit if the number is less than about $6E-5$

A936:lda &30

A938:cmp #&73

A93A:bcc &A904

Pack $FAC\#1$ to &476

A93C:jsr &A381

Zero $FAC\#2$

A93F:jsr &A453

Set $FAC\#2$ to -0.5

A942:lda #&80

A944:sta &3D

A946:sta &3E

A948:sta &3B

Set $FAC\#1 = FAC\#1 + FAC\#2$

A94A:jsr &A505

Call the series routine with the series table at &A95A

A94D:lda #&5A

A94F:ldy #&A9

A951:jsr &A897

Set FAC#1 = FAC#1*476

A954:jsr &AAD1

Indicate that the result is real

A957:lda #&FF

Exit

A959:rts

ATN series table

There are 10 entries in the table

A95A:09-

The first constant is -20.4189003

A95B:85-

A95C:A3-#

A95D:59-Y

A95E:E8-h

A95F:67-g

The second constant is 0.6117710597

A960:80-

A961:1C-

A962:9D-

A963:07-

A964:36-6

The third constant is 0.842704348

A965:80-

A966:57-W

A967:BB-;

A968:78-x

A969:DF-__

The fourth constant is -0.7914132185

A96A:80-

A96B:CA-J

A96C:9A-

A96D:0E-

A96E:83-

The fifth constant is -8.795847349

A96F:84-

A970:8C-

A971:BB-;

A972:CA-J

A973:6E-n

The sixth constant is -1.168640955

A974:81-

A975:95-
A976:96-
A977:06-
A978:DE-^

The seventh constant is 1.084210911

A979:81-
A97A:0A-
A97B:C7-G
A97C:6C-1
A97D:52-R

The eighth constant is 0.4954648205

A97E:7F-
A97F:7D-}
A980:AD--
A981:90-
A982:A1-!

The ninth constant is -3.927877231

A983:82-
A984:FB-{
A985:62-b
A986:57-W
A987:2F-/

The tenth constant is 0.9272952182

A988:80-
A989:6D-m
A98A:63-c
A98B:38-8
A98C:2C-,

COS function routine

Evaluate the operand and ensure it is real

A98D:jsr &92FA

Get the quadrant of the angle

A990:jsr &A9D3

Increment the quadrant. This is because, in graphical terms, the COS curve is merely offset from the SIN curve

A993:inc &4A

Execute the SIN code

A995:jmp &A99E

SIN function routine

Get the argument of the function and ensure it is real

A998:jsr &92FA

Compute the quadrant of the angle

A99B:jsr &A9D3

Skip to &A9AA if the answer should not be negated

A99E:lda &4A

A9A0:and #&2

A9A2:beq &A9AA

Call the main SIN code

A9A4:jsr &A9AA

Negate the result and exit

A9A7:jmp &AD7E

Goto &A9C3 if the result will be less than 0.707 or greater than -0.707 ($\text{SQR}(2)/2$)

A9AA:lsr &4A

A9AC:bcc &A9C3

Call the code used when the result is between -0.707 and 0.707

A9AE:jsr &A9C3

Pack FAC#1 to &46C onwards

A9B1:jsr &A385

Set FAC#1 = FAC#1 * (&4B) (ie FAC#1 = FAC#12)

A9B4:jsr &A656

Pack FAC#1 to &46C onwards

A9B7:jsr &A38D

Set FAC#1 = 1

A9BA:jsr &A699

Set FAC#1 = (&4B) - FAC#1, which is (&46C) - 1 in this case

A9BD:jsr &A4D0

Set FAC#1 = $\text{SQR}(\text{FAC}\#1)$ and exit

A9C0:jmp &A7B7

Save FAC#1 at &476 onwards

A9C3:jsr &A381

Set FAC#1 = FAC#1 * (&4B) (ie FAC#1 = FAC#12)

A9C6:jsr &A656

Evaluate the series at &AA72

A9C9:lda #&72

A9CB:ldy #&AA

A9CD:jsr &A897

Set FAC#1 = FAC#1 * (&476) and exit

A9D0:jmp &AAD1

Find the quadrant of an angle

This routine performs in the same way as this BASIC program:

> LIST

10 REM Filename:D&A9D3

20

30 REM BASIC version of &A9D3 routine

```

40
50 REM (c) 1983 Jeremy Ruston & Acorn
60
70 INPUT A$:FAC1 = EVAL(A$)
80 GOSUB 120
90 PRINT FAC1
100 END
110
120 IF FAC1 >= 2^23 THEN PRINT "Accuracy lost":END
130 S46C = FAC1
140 FAC2 = PI/2
150 S3B = SGN(FAC1)
160 FAC2 = FAC2/2 FAC1 = FAC1 + FAC2
180 FAC1 = FAC1/(PI/2)
190 FAC1 = INT(FAC1)
200 S4A = FAC1 MOD 256
210 IF FAC1 = 0 THEN FAC1 = S46C:RETURN
220 S471 = FAC1
230 FAC1 = FAC1*-1.57080078
240 FAC1 = FAC1 + S46C
250 S46C = FAC1
260 FAC1 = S471
270 FAC1 = FAC1*4.45445511E-6
280 FAC1 = FAC1 + S46C
290 RETURN

```

>

It can be simplified to this program, which makes it a little easier to follow:

> LIST

```

10 REM Filename:D&A9D3
20
30 REM BASIC version of &A9D3 routine
40
50 REM (c) 1983 Jeremy Ruston & Acorn
60
70 @% = &020610
80
90 FOR T = 0 TO PI*2 STEP 0.1
100 FAC1 = T:GOSUB 150
110 PRINT T,FAC1
120 NEXT T
130 END
140
150 S3B = SGN(FAC1)
160 IF FAC1 >= 2^23 THEN PRINT "Accuracy lost":END

```



```

170 S46C = FAC1
180 FAC1 = INT((FAC1 + PI/4)/(PI/2))
190 S4A = FAC1 MOD 256
200 IF FAC1 = 0 THEN FAC1 = S46C:RETURN
210 FAC1 = FAC1*4.45445511E-6 + FAC1*-1.57080078 + S46C
220 RETURN

```

>

Give an 'Accuracy lost' error message if the exponent of the argument is over &97, which means the number is over 8388607

```

A9D3:lda &30
A9D5:cmp #&98
A9D7:bcs &AA38

```

Save the number at &46C onwards

```
A9D9:jsr &A385
```

Make (&4B) point to PI/2

```
A9DC:jsr &AA55
```

Get (&4B) into FAC#2

```
A9DF:jsr &A34E
```

Save the sign of the number

```
A9E2:lda &2E
```

```
A9E4:sta &3B
```

Decrement the exponent of PI/2, turning it into PI/4

```
A9E6:dec &3D
```

Set FAC#1 = FAC#1 + FAC#2

```
A9E8:jsr &A505
```

Set FAC#1 = FAC#1/(PI/2)

```
A9EB:jsr &A6E7
```

Fix the result

```
A9EE:jsr &A3FE
```

Save the least significant byte of the result as the quadrant of the angle

```
A9F1:lda &34
```

```
A9F3:sta &4A
```

Exit, with PI/2 as the result, if the integer part is zero

```
A9F5:ora &33
```

```
A9F7:ora &32
```

```
A9F9:ora &31
```

```
A9FB:beq &AA35
```

Make the exponent be &A0, to turn the integral part into a floating point number

```
A9FD:lda #&A0
```

```
A9FF:sta &30
```

Zero the rounding byte

```
AA01:ldy #&0
```

```
AA03:sty &35
```

Move the sign into the sign byte

AA05:lda &31

AA07:sta &2E

Negate the mantissa if needed

AA09:bpl &AA0E

AA0B:jsr &A46C

Normalise the result

AA0E:jsr &A303

Pack FAC #1 to &471 onwards

AA11:jsr &A37D

Make (&4B) point to &AA59, which contains the constant -1.57080078

AA14:jsr &AA48

Set FAC #1 = FAC #1 * (&4B)

AA17:jsr &A656

Make (&4B) point to &46C

AA1A:jsr &A7F5

Set FAC #1 = FAC #1 + (&4B)

AA1D:jsr &A500

Pack FAC #1 to &46C

AA20:jsr &A38D

Make (&4B) point to &471

AA23:jsr &A7ED

Unpack (&4B) to FAC #1

AA26:jsr &A3B5

Make (&4B) point to &AA5E, which is the constant 4.45445511E-6

AA29:jsr &AA4C

Set FAC #1 = FAC #1 * (&4B)

AA2C:jsr &A656

Make (&4B) point to &46C

AA2F:jsr &A7F5

Set FAC #1 = FAC #1 + (&4B) and exit

AA32:jmp &A500

Unpackk (&4B) into FAC #1 and exit

AA35:jmp &A3B2

AA38:brk

AA39:17-

AA3A:41-A

AA3B:63-c

AA3C:63-c

AA3D:75-u

AA3E:72-r

AA3F:61-a

AA40:63-c

AA41:79-y

AA42:20-

AA43:6C-l
 AA44:6F-o
 AA45:73-s
 AA46:74-t
 AA47:00-

Make (&4B) point to -1.57080078

AA48:lda #&59
 AA4A:bne &AA4E

Make (&4B) point to 4.45445511E-6

AA4C:lda #&5E
 AA4E:sta &4B
 AA50:lda #&AA
 AA52:sta &4C
 AA54:rts

Make (&4B) point to 1.57079633

AA55:lda #&63
 AA57:bne &AA4E

The first constant in this area has been translated as assembly language by the program used to dump the ROM. It is actually the sequence &81 C9100000, which corresponds to the constant -1.57080078

AA59:sta (&C9,X)
 AA5B:bpl &AA5D
 AA5D:brk

This is the constant 4.45445511E-6

AA5E:6F-o
 AA5F:15-
 AA60:77-w
 AA61:7A-z
 AA62:61-a

This is the constant 1.57079633

AA63:81-
 AA64:49-I
 AA65:0F-
 AA66:DA-Z
 AA67:A2-"

This is the constant 1.74532925E-2

AA68:7B-
 AA69:0E-
 AA6A:FA-z

AA6B:35-5

AA6C:12-

This is the constant 57.2957795

AA6D:86-

AA6E:65-e

AA6F:2E-.

AA70:E0-‘

AA71:D3-S

The SIN/COS series table

There are six entries in the table

AA72:05-

The first constant is -8.68214045

AA73:84-

AA74:8A-

AA75:EA-j

AA76:0C-

AA77:1B-

The second is 9.67156522

AA78:84-

AA79:1A-

AA7A:BE->

AA7B:BB-;

AA7C:2B- +

The third is 11.4544274

AA7D:84-

AA7E:37-7

AA7F:45-E

AA80:55-U

AA81:AB- +

The fourth is -3.33333385

AA82:82-

AA83:D5-U

AA84:55-U

AA85:57-W

AA86:7C-|

The fifth is -6.00000001

AA87:83-

AA88:C0-@

AA89:00-

AA8A:00-

AA8B:05-

The sixth is 1

AA8C:81-
AA8D:00-
AA8E:00-
AA8F:00-
AA90:00-

EXP function routine

This is very similar to the exponentiation routine. It evaluates 'e^x' by first working out the result with the fractional part of 'X' and then again with the integral part of 'X', before multiplying the two answers together

Evaluate the argument and ensure it is an integer

AA91:jsr &92FA

Goto &AAB8 if the number is less than 128

AA94:lda &30

AA96:cmp #&87

AA98:bcc &AAB8

Goto &AAA2 if the number is greater than 128

AA9A:bne &AAA2

Jump to &AAB8 if the exponent is less than &B3

AA9C:ldy &31

AA9E:cpy #&B3

AAA0:bcc &AAB8

Give an 'Exp range' error message if the number is positive

AAA2:lda &2E

AAA4:bpl &AAAC

Set the answer to zero

AAA6:jsr &A686

Indicate that the result is real

AAA9:lda #&FF

Exit

AAAB:rts

AAAC:brk

AAAD:18-

AAAE:45-E

AAAF:78-x

AAB0:70-p

AAB1:20-

AAB2:72-r

AAB3:61-a

AAB4:6E-n

AAB5:67-g

AAB6:65-e

AAB7:00-

Set FAC#1 to the fractional part of FAC#1, with the integral part in &4A

AAB8:jsr &A486

Evaluate the series table

AABB:jsr &AADA

Pack FAC#1 to &476

AABE:jsr &A381

Make (&4B) point to &AAE4, which is the constant 'e', or 2.718281828

AAC1:lda #&E4

AAC3:sta &4B

AAC5:lda #&AA

AAC7:sta &4C

Unpack (&4B) to FAC#1

AAC9:jsr &A3B5

Evaluate the integral part of the power

AACC:lda &4A

AACE:jsr &AB12

Make (&4B) point to &476

AAD1:jsr &A7F1

Set FAC#1 = FAC#1 * (&4B)

AAD4:jsr &A656

Indicate that the result is real

AAD7:lda #&FF

Exit

AAD9:rts

Evaluate the series table at &AAE9

AADA:lda #&E9

AADC:ldy #&AA

AADE:jsr &A897

Indicate that the result is real

AAE1:lda #&FF

Exit

AAE3:rts

This is the constant 'e' or 2.718281828

AAE4:82-

AAE5:2D--

AAE6:F8-x

AAE7:54-T

AAE8:58-X

The EXP series table

There are eight entries in the table

AAE9:07-

The first constant is -7.003970316

AAEA:83-

**AAEB:E0-
AAEC:20-
AAED:86-
AAEE:5B-[**

The second is -2.005101137

**AAEF:82-
AAF0:80-
AAF1:53-S
AAF2:93-
AAF3:B8-8**

The third is 5.000003161

**AAF4:83-
AAF5:20-
AAF6:00-
AAF7:06-
AAF8:A1-!**

The fourth is 2.00000796

**AAF9:82-
AAFA:00-
AAFB:00-
AAFC:21-!
AAFD:63-c**

The fifth is -3.000000002

**AAFE:82-
A AFF:C0-@
AB00:00-
AB01:00-
AB02:02-**

The sixth is -2.000000011

**AB03:82-
AB04:80-
AB05:00-
AB06:00-
AB07:0C-**

The seventh is 1

**AB08:81-
AB09:00-
AB0A:00-
AB0B:00-
AB0C:00-**

The eighth is 1

AB0D:81-

AB0E:00-**AB0F:00-****AB10:00-****AB11:00-****Set FAC#1 = FAC#1^A**

This routine sets FAC#1 to a specified integer power. The power is passed as a two's complement number in the accumulator

Place the power in X

AB12:tax

Skip if it is positive

AB13:bpl &AB1E

Decrement the power

AB15:dex

Complement the power. Combined with the DEX instruction, the power has now been negated

AB16:txa**AB17:eor #&FF**

Save the power on the stack

AB19:pha

Set FAC#1 = 1/FAC#1

AB1A:jsr &A6A5

Retrieve the power

AB1D:pla

Save the power on the stack

AB1E:pha

Save FAC#1 at &46C

AB1F:jsr &A385

Set FAC#1 to 1

AB22:jsr &A699

Retrieve the power

AB25:pla

Exit if the power is zero

AB26:beq &AB32

Decrement the power and replace it on the stack

AB28:sec**AB29:sbc #&1****AB2B:pha**

Set FAC#1 = FAC#1*(&4B)

AB2C:jsr &A656

Go back to the start

AB2F:jmp &AB25

Exit

AB32:rts

ADVAL function routine

Evaluate the integer operand

AB33:jsr &92E3

Use the LSB of the result as the channel number

AB36:ldx &2A

Indicate the ADVAL OSBYTE call is required

AB38:lda #&80

Call OSBYTE

AB3A:jsr &FFF4

Get the result into A(lsb) and Y(msb)

AB3D:txa

Transfer the result into the IAC and exit

AB3E:jmp &AEEA**POINT function routine**

Evaluate the X coordinate as an integer

AB41:jsr &92DD

Save it on the stack

AB44:jsr &BD94

Check for a comma

AB47:jsr &8AAE

Evaluate the Y coordinate and check the bracket following it

AB4A:jsr &AE56

Ensure it is an integer

AB4D:jsr &92F0

Save the Y coordinate on the machine stack

AB50:lda &2A**AB52:pha****AB53:lda &2B****AB55:pha**

Pull the X coordinate back into the IAC

AB56:jsr &BDEA

Make the top two bytes be the Y coordinate. This makes the IAC be the correct format for the POINT OSWORD call

AB59:pla**AB5A:sta &2D****AB5C:pla****AB5D:sta &2C**

Point to the IAC with X and Y (Y is zero from the pull IAC routine)

AB5F:ldx #&2A

Indicate the POINT OSWORD call is required

AB61:lda #&9

Call OSWORD

AB63:jsr &FFF1

Get the colour of the point

AB66:lda &2E

Return 'TRUE' if the point is off the screen

AB68:bmi &AB9D

Otherwise, place the accumulator in the IAC and exit

AB6A:jmp &AED8

POS function routine

Get the cursor position using OSBYTE &86

AB6D:lda #&86

AB6F:jsr &FFF4

Place the X coordinate in the IAC and exit

AB72:txa

AB73:jmp &AED8

VPOS function routine

Get the cursor position using OSBYTE &86

AB76:lda #&86

AB78:jsr &FFF4

Place the Y coordinate in the IAC and exit

AB7B:tya

AB7C:jmp &AED8

Check if FAC # 1 is zero

AB7F:jsr &A1DA

Return zero if so

AB82:beq &ABA2

Return 1 if it is positive

AB84:bpl &ABA0

Return -1 if it is negative

AB86:bmi &AB9D

SGN function routine

Evaluate the argument

AB88:jsr &ADEC

Give a 'Type mismatch' error if the result is a string

AB8B:beq &ABE6

Go backwards if the result is a real number

AB8D:bmi &AB7F

Exit if the IAC contains zero

AB8F:lda &2D

AB91:ora &2C

AB93:ora &2B

AB95:ora &2A

AB97:beq &ABA5

Return + 1 if the IAC is positive

AB99:lda &2D

AB9B:bpl &ABA0

Call the 'TRUE' routine to get -1 into the IAC and exit

AB9D:jmp &ACC4

Place 1 in the IAC

ABA0:lda #&1

ABA2:jmp &AED8

Indicate the result is an integer

ABA5:lda #&40

Exit

ABA7:rts

LOG function routine

Call the LN function routine

ABA8:jsr &A7FE

Point to the floating point number at &A869, which is 0.434294482

ABAB:ldy #&69

ABAD:lda #&A8

Multiply the result of the LN function by the constant and exit

ABAF:bne &ABB8

RAD function routine

Evaluate the argument as an operand

ABB1:jsr &92FA

Set (&4B) to point to &AA68, which contains the floating point number 1.74532925E-2

ABB4:ldy #&68

ABB6:lda #&AA

ABB8:sty &4B

ABBA:sta &4C

Multiply the contents of FAC#1 by this constant

ABBC:jsr &A656

Indicate that the result is a floating point number and exit

ABBF:lda #&FF

ABC1:rts

DEG function routine

Evaluate the argument as an integer operand

ABC2:jsr &92FA

Point to the constant at &AA6D, which is 57.2957795

ABC5:ldy #&6D

ABC7:lda #&AA

Multiply FAC#1 by this constant and exit

ABC9:bne &ABB8

PI function routine

Get the value of PI/2 into FAC#1

ABCB:jsr &A8FE

Multiply it by two by incrementing the exponent

ABCE:inc &30

Indicate the result is floating point and exit

ABD0:tay

ABD1:rts

USR function routine

Evaluate the address of the routine as an integer operand

ABD2:jsr &92E3

Load the 6502 registers with the required BASIC variables, and call the routine

ABD5:jsr &8F1E

Place the 6502 registers in the IAC

ABD8:sta &2A

ABDA:stx &2B

ABDC:sty &2C

ABDE:php

ABDF:pla

ABE0:sta &2D

Ensure decimal mode is disabled

ABE2:cld

Indicate that the result is an integer

ABE3:lda #&40

Exit

ABE5:rts

Direct jump to the 'Type mismatch' error message

ABE6:jmp &8C0E

EVAL function routine

The argument to the EVAL function is moved onto the stack before it is evaluated - otherwise, a line such as 'PRINT EVAL("ASC("Hello"))' would not work - think about it

Evaluate the argument as an operand

ABE9:jsr &ADEC

Give a 'Type mismatch' error if the result is not a string

ABEC:bne &ABE6

Add a carriage return onto the end of the string

ABEE:inc &36

ABF0:ldy &36

ABF2:lda #&D

ABF4:sta &5FF,Y

Save the string on the stack

ABF7:jsr &BDB2

Save the location of PTR #2 on the stack

ABFA:lda &19

ABFC:pha

ABFD:lda &1A

ABFF:pha

AC00:lda &1B

AC02:pha

Get the current value of the stack pointer

AC03:ldy &4

AC05:ldx &5

Increment the LSB, to allow for the string length byte on the stack

AC07:iny

Save this address both as PTR #2 and as the tokenising pointer

AC08:sty &19

AC0A:sty &37

Increment the MSB if necessary

AC0C:bne &AC0F

AC0E:inx

Save the MSB as PTR #2 and the tokenising pointer

AC0F:stx &1A

AC11:stx &38

Set the tokenising flag at &3B to &FF to indicate that we are not at the start of a statement

AC13:ldy #&FF

AC15:sty &3B

Zero the offset of PTR #2

AC17:iny

AC18:sty &1B

Tokenise the string

AC1A:jsr &8955

Evaluate the expression at PTR #2

AC1D:jsr &9B29

Discard the string on the stack

AC20:jsr &BDDC

Restore PTR #2

AC23:pla

AC24:sta &1B

AC26:pla

AC27:sta &1A

AC29:pla

AC2A:sta &19

Indicate the correct type

AC2C:lda &27

Exit

AC2E:rts**VAL function routine**

Evaluate the argument as an operand

AC2F:jsr &ADEC

Give a 'Type mismatch' error if the result is not a string

AC32:bne &AC9B

Add a zero to the end of the string, to indicate the end of the string to the ASCII to binary conversion routine

AC34:ldy &36

AC36:lda #&0

AC38:sta &600,Y

Save PTR #2 on the machine stack

AC3B:lda &19

AC3D:pha

AC3E:lda &1A

AC40:pha

AC41:lda &1B

AC43:pha

Zero the offset

AC44:lda #&0

AC46:sta &1B

Point to the string buffer with the pointer itself

AC48:lda #&0

AC4A:sta &19

AC4C:lda #&6

AC4E:sta &1A

Get the next character

AC50:jsr &8A8C

Jump forwards if it is a minus sign

AC53:cmp #&2D

AC55:beq &AC66

Skip the plus sign, if present

AC57:cmp #&2B

AC59:bne &AC5E

AC5B:jsr &8A8C

AC5E:dec &1B

Get a number from the ASCII

AC60:jsr &A07B

Exit via the end of the EVAL routine to restore PTR #2

AC63:jmp &AC73

Skip to the first digit of the number

AC66:jsr &8A8C

Back the pointer over the digit

AC69:dec &1B

Get a number from the ASCII

AC6B:jsr &A07B

Negate the number, if one was found

AC6E:bcc &AC73

AC70:jsr &AD8F

Save the type

AC73:sta &27

Exit via the end of the 'EVAL' code

AC75:jmp &AC23

INT function routine

Evaluate the argument of the function

AC78:jsr &ADEC

Give a 'Type mismatch' error if the result is a string

AC7B:beq &AC9B

Exit if the result is a integer

AC7D:bpl &AC9A

Save the sign of the floating point number

AC7F:lda &2E

AC81:php

Remove the fractional part of the number

AC82:jsr &A3FE

Simply move the number into the IAC and exit if the number is positive

AC85:plp

AC86:bpl &AC95

Or is zero

AC88:lda &3E

AC8A:ora &3F

AC8C:ora &40

AC8E:ora &41

AC90:beq &AC95

Otherwise, negate it

AC92:jsr &A4C7

Move the number into the IAC

AC95:jsr &A3E7

Indicate the result is an integer

AC98:lda #&40

Exit

AC9A:rts

Direct jump to the 'Type mismatch' error

AC9B:jmp &8C0E

ASC function routine

Evaluate the argument to the function as an operand

AC9E:jsr &ADEC

Give a 'Type mismatch' error if the result is not a string

ACA1:bne &AC9B

Return -1 if the string is null

ACA3:lda &36

ACA5:beq &ACC4

Otherwise, place the first character of the string into the IAC and exit

ACA7:lda &600

ACAA:jmp &AED8

INKEY function routine

Call the INKEY master routine

ACAD:jsr &AFAD

Return -1 if no character was recorded

ACB0:cpy #&0

ACB2:bne &ACC4

Otherwise, place the character in the IAC and exit

ACB4:txa

ACB5:jmp &AEEA

EOF function routine

Evaluate the handle of the file

ACB8:jsr &BFB5

Get the file handle into X

ACBB:tax

Use OSBYTE &7F to find out if the file has ended

ACBC:lda #&7F

ACBE:jsr &FFF4

Return zero if the file has not finished

ACC1:txa

ACC2:beq &ACAA

Otherwise, fall into the 'TRUE' routine

TRUE function routine

Place &FF into all the bytes of the IAC

ACC4:lda #&FF

ACC6:sta &2A

ACC8:sta &2B

ACCA:sta &2C

ACCC:sta &2D

Indicate the result is an integer

ACCE:lda #&40

Exit

ACD0:rts

NOT function routine

Evaluate the argument as an integer operand

ACD1:jsr &92E3

Invert all the bytes

ACD4:ldx #&3

ACD6:lda &2A,X

ACD8:eor #&FF

ACDA:sta &2A,X

ACDC:dex

ACDD:bpl &ACD6

Indicate the result is an integer

ACDF:lda #&40

Exit

ACE1:rts

INSTR function routine

In this description, the first string is referred to as the search string while the second is the searched string

Evaluate the searched string

ACE2:jsr &9B29

Give a 'Type mismatch' error if the result is not a string

ACE5:bne &AC9B

Give a 'Missing ,' error if the next character is not a comma

ACE7:cpx #&2C

ACE9:bne &AD03

Increment PTR #2 past the comma

ACEB:inc &1B

Push the searched string on the stack

ACED:jsr &BDB2

Evaluate the search string

ACF0:jsr &9B29

Give a 'Type mismatch' error if the result is not a string

ACF3:bne &AC9B

Set the default start position for the string

ACF5:lda #&1

ACF7:sta &2A

Increment the pointer to the next character

ACF9:inc &1B

Skip forwards if the next character is a ')', indicating the default start position should be used

ACFB:cpx #&29

ACFD:beq &AD12

Give a 'Missing ,' error message if the next character is not a comma

ACFF:cpx #&2C

AD01:beq &AD06

AD03: jmp &8AA2

Push the search string on the stack

AD06: jsr &BDB2

Evaluate the start position and check for the right hand bracket following it

AD09: jsr &AE56

Ensure the result is an integer

AD0C: jsr &92F0

Pull the search string back

AD0F: jsr &BDCB

Zero Y for later use

AD12: ldy #&0

If the start position is given as zero, change it to one

AD14: ldx &2A

AD16: bne &AD1A

AD18: ldx #&1

AD1A: stx &2A

Place the start position in A

AD1C: txa

Decrement the start position, to make it a proper offset

AD1D: dex

Save the modified start position in an unused part of the IAC

AD1E: stx &2D

Make (&37) point to the start position in the searched string

AD20: clc

AD21: adc &4

AD23: sta &37

AD25: tya

AD26: adc &5

AD28: sta &38

Get the length of the searched string

AD2A: lda (&4),Y

Subtract the modified start position from the length

AD2C: sec

AD2D: sbc &2D

Exit with zero as the result if the start position is past the end of the searched string

AD2F: bcc &AD52

Exit with zero as the result if the search string could not fit between the start position and the end of the searched string

AD31: sbc &36

AD33: bcc &AD52

Increment this number and save it as the number of search positions left

AD35: adc #&0

AD37: sta &2B

Discard the searched string

AD39: jsr &BDDC

Start searching at the start of the search string

AD3C:ldy #&0

Use X as a counter of the number of characters to be scanned

AD3E:ldx &36

Skip the search if there are no characters to be scanned

AD40:beq &AD4D

Compare a character from the searched string with one in the search string

AD42:lda (&37),Y

AD44:cmp &600,Y

Jump forwards if there is no match

AD47:bne &AD59

Increment the pointer to compare the next characters

AD49:iny

Decrement the number of characters left to inspect

AD4A:dex

If there are still some remaining, continue the search

AD4B:bne &AD42

Place the current search position in the IAC and exit

AD4D:lda &2A

AD4F:jmp &AED8

Discard the searched string

AD52:jsr &BDDC

Return with zero as the result

AD55:lda #&0

AD57:beq &AD4F

Increment the current search position

AD59:inc &2A

Decrement the number of search positions left

AD5B:dec &2B

Exit with zero as the result if we have run out of search positions

AD5D:beq &AD55

Increment the pointer into the searched string and continue the search

AD5F:inc &37

AD61:bne &AD3C

AD63:inc &38

AD65:bne &AD3C

Direct jump to the 'Type mismatch' error

AD67:jmp &8C0E

ABS function routine

Evaluate the argument of the function

AD6A:jsr &ADEC

Give a 'Type mismatch' error if it is a string

AD6D:beq &AD67

Jump forwards if it is a real number

AD6F:bmi &AD77

If the IAC is positive, exit, otherwise, negate it and exit

AD71:bit &2D

AD73:bmi &AD93

AD75:bpl &ADAA

Check whether FAC#1 is zero

AD77:jsr &A1DA

Exit if it is positive

AD7A:bpl &AD89

Invert the sign bit if it is negative

AD7C:bmi &AD83

Check whether FAC#1 is zero

AD7E:jsr &A1DA

Exit if it is

AD81:beq &AD89

Invert the sign bit

AD83:lda &2E

AD85:eor #&80

AD87:sta &2E

Indicate the result is a floating point number

AD89:lda #&FF

Exit

AD8B:rts

Unary '-' routine

Evaluate the quantity after the minus sign

AD8C:jsr &AE02

Give a 'Type mismatch' error if the result was a string

AD8F:beq &AD67

If it was real, jump into the code that dealt with the ABS routine to do the negation

AD91:bmi &AD7E

Set the carry flag in preparation for the subtraction operation

AD93:sec

Prime the accumulator and the Y register with zero. This means that whenever a 'LDA #0' instruction is required, 'TYA' can be used instead, saving one byte and quite a bit of time

AD94:lda #&0

AD96:tay

Negate the first byte

AD97:sub &2A

AD99:sta &2A

Negate the second byte

AD9B:tya

AD9C:sub &2B**AD9E:sta &2B**

Negate the third byte

ADA0:tya**ADA1:sub &2C****ADA3:sta &2C**

Negate the fourth byte

ADA5:tya**ADA6:sub &2D****ADA8:sta &2D**

Indicate that the result is an integer

ADAA:lda #&40

Exit

ADAC:rts**Get string into string buffer**

This routine is used by the READ and INPUT statements to get a string into the buffer. The string may or may not be surrounded by quotes - if it is, the next routine is used, otherwise this routine is used

Get the next character

ADAD:jsr &8A8C

Join the quoted string routine if it is a quote

ADB0:cmp #&22**ADB2:beq &ADC9**

Indicate that the first location in the string buffer is to be used next

ADB4:ldx #&0

Get a character from the text

ADB6:lda (&19),Y

Save it in the buffer

ADB8:sta &600,X

Increment the pointers

ADBB:iny**ADBC:inx**

Exit if the character is a carriage return

ABD:cmp #&D**ABDF:beq &ADC5**

Continue as long as the character is not a comma

ADC1:cmp #&2C**ADC3:bne &ADB6**

Decrement the offset past the comma or the carriage return

ADC5:dey

Exit through the end of the quoted string routine

ADC6:jmp &ADE1

Place quoted string in string buffer

The quoted string is at PTR #2

Zero the pointer into the string buffer

ADC9:ldx #&0

Get the next character

ADCB:iny

ADCC:lda (&19),Y

Give a 'Missing "' error message if the character is a carriage return

ADCE:cmp #&D

ADD0:beq &ADE9

Point to the next source character

ADD2:iny

Save the character in the buffer

ADD3:sta &600,X

Point to the next location in the buffer

ADD6:inx

If the character is not a double quotation mark, go back and get another character

ADD7:cmp #&22

ADD9:bne &ADCC

Get the next character

ADDB:lda (&19),Y

Continue the search if another quotation mark follows it - notice that the first quotation mark will have been placed in the buffer

ADDD:cmp #&22

ADDF:beq &ADCB

Decrement the length of the string to remove the final quote symbol

ADE1:dex

Save the length of the string

ADE2:stx &36

Update the offset of PTR #2

ADE4:sty &1B

Indicate that the result is a string

ADE6:lda #&0

Exit

ADE8:rts

Direct jump to the 'Missing "' error message

ADE9:jmp &8E98

Evaluate operand routine

This routine evaluates an operand at PTR #2. It is the final destination of the expression evaluation routine, and it is used by functions to evaluate their arguments

Get the next character

ADEC:ldy &1B

ADEE:inc &1B

```

ADF0:lda (&19),Y
Get a new character if it is a space
ADF2:cmp #&20
ADF4:beq &ADEC
Join the unary minus sign routine if the character is a minus sign
ADF6:cmp #&2D
ADF8:beq &AD8C
Join the quoted string routine if the character is a quote
ADFA:cmp #&22
ADFC:beq &ADC9
Skip the character if it is a plus sign
ADFE:cmp #&2B
AE00:bne &AE05
AE02:jsr &8A8C
Goto &AE10 if the character is too low to be a function token
AE05:cmp #&8E
AE07:bcc &AE10
Give a 'No such variable message' if the character is too high to be a token for a function
AE09:cmp #&C6
AE0B:bcs &AE43
Execute the required function by jumping to its action address
AE0D:jmp &8BB1

Goto &AE20 if the character is greater than or equal to &3F
AE10:cmp #&3F
AE12:bcs &AE20
Goto &AE2A if the character is greater than or equal to &2E
AE14:cmp #&2E
AE16:bcs &AE2A
Goto &AE6D if the character is a '&'
AE18:cmp #&26
AE1A:beq &AE6D
Goto &AE56 if the character is a '('
AE1C:cmp #&28
AE1E:beq &AE56
Decrement the pointer past the character
AE20:dec &1B
Get a variable name
AE22:jsr &95DD
Exit if it does not exist
AE25:beq &AE30
Exit, getting the value of the variable
AE27:jmp &B32C

Get a number from the text
AE2A:jsr &A07B

```

Give a 'No such variable' error if the number does not exist

AE2D:bcc &AE43

Exit

AE2F:rts

Give a 'No such variable' error message if the variable is not valid, or does not exist. If OPT indicates that errors should be suppressed and the variable is numeric, then give the variable the value of P% instead of giving the error message

AE30:lda &28

AE32:and #&2

AE34:bne &AE43

AE36:bcs &AE43

AE38:stx &1B

Get P% into A and Y

AE3A:lda &440

AE3D:ldy &441

Place A and Y into the IAC and exit

AE40:jmp &AEEA

AE43:brk

AE44:1A-

AE45:4E-N

AE46:6F-o

AE47:20-

AE48:73-s

AE49:75-u

AE4A:63-c

AE4B:68-h

AE4C:20-

AE4D:76-v

AE4E:61-a

AE4F:72-r

AE50:69-i

AE51:61-a

AE52:62-b

AE53:6C-l

AE54:65-e

AE55:00-

Deal with a bracketed expression

Evaluate the expression in the brackets

AE56:jsr &9B29

Increment the pointer past the alleged ')'

AE59:inc &1B

Give a 'Missing)' error message if the next character is not a ')'

AE5B:cpx #&29

AE5D:bne &AE61

Exit

AE5F:tay

AE60:rts

AE61:brk

AE62:1B-

AE63:4D-M

AE64:69-i

AE65:73-s

AE66:73-s

AE67:69-i

AE68:6E-n

AE69:67-g

AE6A:20-

AE6B:29-)

AE6C:00-

Decode a hexadecimal number

This routine is entered with PTR #2 pointing at the first character of a hex number

Zero the IAC

AE6D:ldx #&0

AE6F:stx &2A

AE71:stx &2B

AE73:stx &2C

AE75:stx &2D

Get the current character

AE77:ldy &1B

AE79:lda (&19),Y

Exit if the character is too small to be a digit

AE7B:cmp #&30

AE7D:bcc &AEA2

Continue if the character is a digit

AE7F:cmp #&3A

AE81:bcc &AE8D

Take off the ASCII factor to turn alphabetical characters into hex digits

AE83:sbc #&37

Exit if the result is less than 10

AE85:cmp #&A

AE87:bcc &AEA2

Exit if the result is 16 or greater

AE89:cmp #&10

AE8B:bcs &AEA2

Get the digit into the upper nybble of A

AE8D:asl A

AE8E:asl A**AE8F:asl A****AE90:asl A**

Ripple the nybble into the IAC

AE91:ldx #&3**AE93:asl A****AE94:rol &2A****AE96:rol &2B****AE98:rol &2C****AE9A:rol &2D****AE9C:dex****AE9D:bpl &AE93**

Point to the next character, and continue scanning the digits

AE9F:iny**AEA0:bne &AE79**

Give a 'Bad HEX' message if the character which caused conversion to stop was the first character

AEA2:txa**AEA3:bpl &AEAA**

Otherwise, update the offset of PTR#2

AEA5:sty &1B

Indicate that the result is an integer

AEA7:lda #&40

Exit

AEA9:rts**AEAA:brk****AEAB:1C-****AEAC:42-B****AEAD:61-a****AEAE:64-d****AEAF:20-****AEB0:48-H****AEB1:45-E****AEB2:58-X****AEB3:00-****TIME function routine**

Point to the IAC with X and Y

AEB4:ldx #&2A**AEB6:ldy #&0**

Use OSWORD &01 to read the time

AEB8:lda #&1**AEBA:jsr &FFF1**

Indicate the result is an integer

AEBD:lda #&40

Exit

AEBF:rts

PAGE function routine

Set A and Y to contain PAGE

AEC0:lda #&0

AEC2:ldy &18

Place this value in the IAC and exit

AEC4:jmp &AEEA

Direct jump to the 'No such variable' error message

AEC7:jmp &AE43

FALSE function routine

Set A to zero

AECA:lda #&0

Exit, placing A in the IAC

AECC:beq &AED8

Direct jump to the 'Type mismatch' error

AECE:jmp &8C0E

LEN function routine

Evaluate the argument

AED1:jsr &ADEC

Give a 'Type mismatch' error if the result is not a string

AED4:bne &AECE

Get the length of the string into A

AED6:lda &36

Set the MSB of the length to zero

AED8:ldy #&0

Place A and Y in the IAC and exit

AEDA:beq &AEEA

TO(P) function routine

TOP does not have a token of its own; the token or 'TO' (as in 'FOR') is used, and a check is made for a letter 'P' following the token

Get the next character

AEDC:ldy &1B

AEDE:lda (&19),Y

AEE0:cmp #&50

Give a 'No such variable' message if the 'P' is not present

AEE2:bne &AEC7

Increment PTR #2 past the 'P'

AEE4:inc &1B

Set A and Y to contain TOP

AEE6:lda &12

AEE8:ldy &13

Place this value in the IAC

AEEA:sta &2A

AEEC:sty &2B

Zero the top two bytes of the IAC

AEEE:lda #&0

AEF0:sta &2C

AEF2:sta &2D

Indicate that the result is an integer

AEF4:lda #&40

Exit

AEF6:rts

COUNT function routine

Get the current value of COUNT

AEF7:lda &1E

Exit, placing it in the IAC

AEF9:jmp &AED8

LOMEM function routine

Get LOMEM into A and Y

AEFC:lda &0

AEFE:ldy &1

Exit, placing LOMEM in the IAC

AF00:jmp &AEEA

HIMEM function routine

Get HIMEM into A and Y

AF03:lda &6

AF05:ldy &7

Exit, placing HIMEM in the IAC

AF07:jmp &AEEA

Increment PTR #2 past the opening bracket

AF0A:inc &1B

Evaluate the expression following, and check the presence of the right hand bracket

AF0C:jsr &AE56

Ensure the result of the expression is an integer

AF0F:jsr &92F0

Goto &AF3F if the integer is negative

AF12:lda &2D

AF14:bmi &AF3F

Check if the argument is zero

AF16:ora &2C

AF18:ora &2B

Goto &AF24 if it is not
 AF1A:bne &AF24
 Goto &AF6C if the argument is zero
 AF1C:lda &2A
 AF1E:beq &AF6C
 Goto &AF69 if the argument is one
 AF20:cmp #&1
 AF22:beq &AF69

RND(+ X) entry

Convert the limit to a real number
 AF24:jsr &A2BE
 Save the limit on the stack
 AF27:jsr &BD51
 Get a random number between 0 and 1
 AF2A:jsr &AF69
 Discard the limit from the stack, leaving (&4B) pointing to it
 AF2D:jsr &BD7E
 Multiply the random number by the limit
 AF30:jsr &A606
 Normalise the result
 AF33:jsr &A303
 Convert the result back to an integer
 AF36:jsr &A3E4
 Increment the number generated - zero is not allowed to be returned
 AF39:jsr &9222
 Indicate that the result is an integer and exit
 AF3C:lda #&40
 AF3E:rts

RND(-X) entry

Copy the IAC to the RND store
 AF3F:ldx #&D
 AF41:jsr &BE44
 Indicate that the result is an integer
 AF44:lda #&40
 Set the top byte of the store as well
 AF46:sta &11
 Exit
 AF48:rts

RND function entry

The only interesting point to notice is that functions like 'RND(10)' are computed using floating point arithmetic, so it is a better idea to use the faster 'RND MOD X' form to get a random number between 0 and X-1

Get the next character

AF49:ldy &1B

AF4B:lda (&19),Y

Go backwards if it is a '(', which indicates that an argument is being used

AF4D:cmp #&28

AF4F:beq &AF0A

Generate a new random seed

AF51:jsr &AF87

Indicate that the number starts at location &0D

AF54:ldx #&D

Fall into next routine

Copy &0X onwards to the IAC

AF56:lda &0,X

AF58:sta &2A

AF5A:lda &1,X

AF5C:sta &2B

AF5E:lda &2,X

AF60:sta &2C

AF62:lda &3,X

AF64:sta &2D

Indicate that the result is an integer

AF66:lda #&40

Exit

AF68:rts

RND(1) entry

Update the RND seed

AF69:jsr &AF87

RND(0) entry

Zero the sign, overflow and rounding bytes

AF6C:ldx #&0

AF6E:stx &2E

AF70:stx &2F

AF72:stx &35

Set the exponent to &80, to ensure the number is between 0 and 1

AF74:lda #&80

AF76:sta &30

Copy the seed into the mantissa of FAC#1

AF78:lda &D,X

AF7A:sta &31,X

AF7C:inx

AF7D:cpx #&4

AF7F:bne &AF78

Normalise the result

AF81:jsr &A659

Indicate that the result is a floating point number

AF84:lda #&FF

Exit

AF86:rts

Update the RND seed

This routine generates a new random number from the old one using the shift register technique. Using this technique, the same random number comes up once every 233-1 times

Do 32 iterations

AF87:ldy #&20

Get bit 4 of location &0F into bit 0 of A

AF89:lda &F

AF8B:lsr A

AF8C:lsr A

AF8D:lsr A

Exclusive-OR this result with the value in location &11

AF8E:eor &11

Ripple this value throughout the store

AF90:ror A

AF91:rol &D

AF93:rol &E

AF95:rol &F

AF97:rol &10

AF99:rol &11

Continue for the remaining iterations

AF9B:dey

AF9C:bne &AF89

Exit

AF9E:rts

ERL function routine

Get the line number of the most recent error into A and Y

AF9F:ldy &9

AFA1:lda &8

Place it in the IAC and exit

AFA3:jmp &AEEA

ERR function routine

Get zero into Y

AFA6:ldy #&0

Get the number of the last error into A

AFA8:lda (&FD),Y

Place the number in the IAC and exit

AFAA:jmp &AEEA

Basic INKEY routine

Evaluate the time limit

AFAD:jsr &92E3

Use OSBYTE &81

AFB0:lda #&81

Get the delay into X and Y

AFB2:ldx &2A

AFB4:ldy &2B

Call OSBYTE

AFB6:jmp &FFF4

GET function routine

Get a character

AFB9:jsr &FFE0

Place it in the IAC and exit

AFBC:jmp &AED8

GET\$ function routine

Get a character

AFBF:jsr &FFE0

Save it as a string of length 1

AFC2:sta &600

AFC5:lda #&1

AFC7:sta &36

Indicate that the result is a string

AFC9:lda #&0

Exit

AFCB:rts

LEFT\$ function routine

Evaluate the source string

AFCC:jsr &9B29

Give a 'Type mismatch' error if the result is not a string

AFCF:bne &B033

Give a 'Missing , ' error message if the next character is not a comma

AFD1:cpx #&2C

AFD3:bne &B036

Increment PTR #2 past the comma

AFD5:inc &1B

Push the string on the stack

AFD7:jsr &BDB2

Evaluate the length and check for the right hand bracket

AFDA:jsr &AE56

Ensure the length is an integer

AFDD:jsr &92F0

Pull the string

AFE0:jsr &BDCB

Compare the length of the string with the required length

AFE3:lda &2A

AFE5:cmp &36

Exit immediately if the required length is too big

AFE7:bcs &AFEB

Change the length of the string to the required length

AFE9:sta &36

Indicate that the result is a string

AFEB:lda #&0

Exit

AFED:rts

RIGHT\$ function routine

Evaluate the source string

AFEE:jsr &9B29

Give a 'Type mismatch' error if the result is not a string

AFF1:bne &B033

Give a 'Missing ,' error message if the next character is not a comma

AFF3:cpx #&2C

AFF5:bne &B036

Increment PTR #2 past the comma

AFF7:inc &1B

Save the string on the stack

AFF9:jsr &BDB2

Evaluate the length and check for the closing bracket

AFFC:jsr &AE56

Ensure the length is an integer

FFFF:jsr &92F0

Pull the string from the stack

B002:jsr &BDCB

Subtract the required length from the actual length

B005:lda &36

B007:sec

B008:sbc &2A

If the requested length is too big, exit

B00A:bcc &B023

If the string is the requested length, exit

B00C:beq &B025

Save the offset of the start of the required string in X

B00E:tax

Set the length of the string to the required length

B00F:lda &2A

B011:sta &36

Exit if the required length is zero

B013:beq &B025

Start by moving characters to the first location in the buffer

B015:ldy #&0

Get a character from X

B017:lda &600,X

Place it further down at Y

B01A:sta &600,Y

Increment the pointers

B01D:inx

B01E:iny

Decrement the required length

B01F:dec &2A

Continue if the required length has not been met

B021:bne &B017

Indicate that the result is a string

B023:lda #&0

Exit

B025:rts

INKEY\$ function routine

Call the basic INKEY routine

B026:jsr &AFAD

Save the character in A

B029:txa

Return the character pressed if the time limit was not exceeded

B02A:cpy #&0

B02C:beq &AFC2

Give the null string as the result, as no key was pressed

Return the null string

B02E:lda #&0

B030:sta &36

Exit

B032:rts

Direct jump to the 'Type mismatch' error message

B033:jmp &8C0E

Direct jump to the 'Missing ,' error message

B036:jmp &8AA2

MID\$ function routine

Evaluate the string

B039:jsr &9B29

Give a 'Type mismatch' error if the result is not a string

B03C:bne &B033

Give a 'Missing ,' error message if the next character is not a comma

B03E:cpx #&2C**B040:bne &B036**

Save the string on the stack

B042:jsr &BDB2

Increment PTR #2 past the comma

B045:inc &1B

Evaluate the starting position

B047:jsr &92DD

Save it on the machine stack

B04A:lda &2A**B04C:pha**

Set the length to &FF as the default

B04D:lda #&FF**B04F:sta &2A**

Increment past the next character

B051:inc &1B

Skip evaluating the length if the next character is a ')'

B053:cpx #&29**B055:beq &B061**

Give a 'Missing ,' error message if the next character is not a comma

B057:cpx #&2C**B059:bne &B036**

Evaluate the length and the closing bracket

B05B:jsr &AE56

Ensure the length is an integer

B05E:jsr &92F0

Get the string back off the stack

B061:jsr &BDCB

Save the starting location in Y

B064:pla**B065:tay**

If the start location is given as zero, jump forwards

B066:clc**B067:beq &B06F**

Take off the length of the string

B069:sbc &36

If the start location is past the end of the string, return the null string

B06B:bcs &B02E

Decrement Y to make it into a proper offset

B06D:dey

Place it in A

B06E:tya

Save the start location in &2C

B06F:sta &2C

Place it in X

B071:tax

Zero Y

B072:ldy #&0

Take the starting location away from the length of the string

B074:lda &36

B076:sec

B077:sub &2C

Compare the result with the number of characters requested

B079:cmp &2A

If the requested length is too big, use the remaining length of the string as the requested length

B07B:bcs &B07F

B07D:sta &2A

Get the length of the substring

B07F:lda &2A

Return with the null string if the substring has a length of zero

B081:beq &B02E

Move the characters down

B083:lda &600,X

B086:sta &600,Y

Increment the pointers

B089:iny

B08A:inx

Continue until enough characters have been moved

B08B:cpy &2A

B08D:bne &B083

Set the length of the substring

B08F:sty &36

Indicate that the result is a string

B091:lda #&0

Exit

B093:rts

STR\$ function routine

Get the next character

B094:jsr &8A8C

Set Y to &FF

B097:ldy #&FF

If the character is a '~', indicating a hex string should be built up, go forwards

B099:cmp #&7E

B09B:beq &B0A1

Set Y to &00, to indicate decimal mode

B09D:ldy #&0

Decrement PTR#2 to allow for not finding a '~'

B09F:dec &1B

Place the flag in A

B0A1:tya

Save it on the stack

B0A2:pha

Evaluate the argument

B0A3:jsr &ADEC

Give a 'Type mismatch' error if the result is a string

B0A6:beq &B0BF

Save the type of the result in Y

B0A8:tay

Save the decimal/hex flag in &15

B0A9:pla

B0AA:sta &15

See if @% should be taken into account

B0AC:lda &403

B0AF:bne &B0B9

If not, zero location &37 and convert the number to a string

B0B1:sta &37

B0B3:jsr &9EF9

Indicate that the result is a string

B0B6:lda #&0

Exit

B0B8:rts

Convert the number to a string

B0B9:jsr &9EDF

Indicate that the result is a string

B0BC:lda #&0

Exit

B0BE:rts

Direct jump to the 'Type mismatch' error message

B0BF:jmp &8C0E

STRING\$ function routine

Evaluate the number of repetitions

B0C2:jsr &92DD

Save the number of repetitions

B0C5:jsr &BD94

Check for a comma

B0C8:jsr &8AAE

Evaluate the string and check for a ')

B0CB:jsr &AE56

Give a 'Type mismatch' error if the result is not a string

B0CE:bne &B0BF

Pull the number of repetitions back

B0D0:jsr &BDEA

Exit if the string is null

B0D3:ldy &36

B0D5:beq &B0F5

Exit with the null string if the number of repetitions is zero

B0D7:lda &2A

B0D9:beq &B0F8

Decrement the number of repetitions so it can be used as a counter

B0DB:dec &2A

Exit with one repetition of the string if only one was requested

B0DD:beq &B0F5

Make a copy of the string and place it immediately after the original string in the buffer

B0DF:ldx #&0

B0E1:lda &600,X

B0E4:sta &600,Y

Increment the pointers

B0E7:inx

B0E8:iny

Exit if the string length has been exceeded

B0E9:beq &B0FB

Continue until the end of the string has been encountered

B0EB:cpx &36

B0ED:bcc &B0E1

Continue the process for the correct number of repetitions

B0EF:dec &2A

B0F1:bne &B0DF

Save the length of the string

B0F3:sty &36

Indicate that the result is a string

B0F5:lda #&0

Exit

B0F7:rts

Save the length of the string, which will be zero

B0F8:sta &36

Exit

B0FA:rts

Direct jump to the 'String too long' error message

B0FB:jmp &9C03

Give a 'No such PROC/FN' error

Retrieve the settings of PTR # 1

```

B0FE:pla
B0FF:sta &C
B101:pla
B102:sta &B

B104:brk
B105:1D-
B106:4E-N
B107:6F-o
B108:20-
B109:73-s
B10A:75-u
B10B:63-c
B10C:68-h
B10D:20-
B10E:A4-$
B10F:2F-/
B110:F2-r
B111:00-

```

Find a PROC/FN not in the catalogue

This routine searches for a PROC/FN in memory if it was not found in the catalogue. It also constructs a catalogue entry for the PROC/FN once it is found

Set PTR # 1 to PAGE

```

B112:lda &18
B114:sta &C
B116:lda #&0
B118:sta &B

```

Get the MSB of the current line number

```

B11A:ldy #&1
B11C:lda (&B),Y

```

If the top bit is set we have reached the end of the program without finding the right PROC/FN, so give a 'No such PROC/FN' error message

```

B11E:bmi &B0FE

```

Point to the length of the line

```

B120:ldy #&3

```

Point to the next character

```

B122:iny

```

Get the character

```

B123:lda (&B),Y

```

Repeat the process if it is a space

```

B125:cmp #&20
B127:beq &B122

```


Jump out of the loop if it is the word 'DEF'

B129:cmp #&DD

B12B:beq &B13C

Get the length of the line

B12D:ldy #&3

B12F:lda (&B),Y

Add the length of the line to PTR# 1, to make it point to the next line, and repeat the search on the new line

B131:clc

B132:adc &B

B134:sta &B

B136:bcc &B11A

B138:inc &C

B13A:bcs &B11A

Save the offset of the character after the word 'DEF'

B13C:iny

B13D:sty &A

Get the next character

B13F:jsr &8A97

Save the offset of the next character in X and A

B142:tya

B143:tax

Add the offset to PTR# 1, to make it point directly to the word 'PROC' or 'FN'. Leave the result in A(lsb) and Y(msb)

B144:clc

B145:adc &B

B147:ldy &C

B149:bcc &B14D

B14B:iny

B14C:clc

Decrement this value and use it to construct a pointer at &3C, &3D

B14D:sbc #&0

B14F:sta &3C

B151:tya

B152:sbc #&0

B154:sta &3D

Initialise the pointer

B156:ldy #&0

Increment the pointer

B158:iny

Increment X, which is keeping track of the offset of the end of the name from the start of the line

B159:inx

Compare a character from the program text to the corresponding character from the name being sought

B15A:lda (&3C),Y

B15C:cmp (&37),Y

Continue the search at the next line if no match is found

B15E:bne &B12D

Continue until the end of the sought name is reached

B160:cpy &39

B162:bne &B158

Get the next character from the program text

B164:iny

B165:lda (&3C),Y

See if it is alphanumeric

B167:jsr &8926

If it is, the definition being scanned is of a PROC/FN with a longer name than the one being sought, so the search must be aborted

B16A:bcs &B12D

Get the offset of the end of the name into Y

B16C:txa

B16D:tay

Add this to PTR # 1

B16E:jsr &986D

Create a catalogue entry for the PROC/FN

B171:jsr &94ED

Clear the byte after the name in the catalogue

B174:ldx #&1

B176:jsr &9531

Store the address of the PROC/FN

B179:ldy #&0

B17B:lda &B

B17D:sta (&2),Y

B17F:iny

B180:lda &C

B182:sta (&2),Y

Update VARTOP

B184:jsr &9539

Join the PROC/FN code

B187:jmp &B1F4

B18A:brk

B18B:1E-

B18C:42-B

B18D:61-a

B18E:64-d

B18F:20-

B190:63-c

B191:61-a

B192:6C-l

FN function routine

Refer to the separate section on procedures and functions for an overview of the way this routine operates

Because this description has to deal with two separate PROC/FN parameters areas, the parameter area of the definition will be referred to as the defined parameter area, while the parameter area of the calling sequence for the PROC/FN will be referred to as the calling parameter area

Indicate that a function is being called, by loading the accumulator with the token for 'FN'. This will be referred to as the type of the call

B195:lda #&A4

Save the type of the call

B197:sta &27

Get the value of the machine stack pointer into X and A

B199:tsx

B19A:txa

Lower the BASIC stack by one more than the number of bytes on the stack. The stack pointer gives the number of items as a one's complement number - thus, when the stack pointer contains &FF, there are no bytes on the stack - so an ADC instruction is needed to lower the BASIC stack

B19B:clc

B19C:adc &4

B19E:jsr &BE2E

Place the size of the machine stack on the BASIC stack

B1A1:ldy #&0

B1A3:txa

B1A4:sta (&4),Y

Make X point to the next byte on the machine stack

B1A6:inx

Make Y point to the next free byte on the BASIC stack

B1A7:iny

Load a byte from the machine stack and save it on the BASIC stack

B1A8:lda &100,X

B1AB:sta (&4),Y

Continue the process until X becomes &FF, indicating that the byte at the top of the machine stack has been transferred

B1AD:cpx #&FF

B1AF:bne &B1A6

Set the machine stack pointer to &FF

B1B1:txs

Save the type of the call on the machine stack. Thus, location &1FF will contain the type of the current procedure or function. Using this information, you can even write a hybrid procedure and function like this:


```

1000 DEF PROCname(X)
1010 DEF FNname(X)
1020 X = X-1 + SIN(Y*2)/COS(RAD(T + FNa(B)))
1030 IF ?&1FF = &A2 THEN = X
1040 PRINT X
1050 ENDPROC

```

The above routine can be called as a procedure or a function. If called as a procedure, the result is printed, otherwise, it is returned as a value. Of course, it would be relatively simple to write the definition without using the machine stack, but the above solution may be superior in some cases

```
B1B2:lda &27
```

```
B1B4:pha
```

Save the current setting of PTR #1 on the machine stack. This is done because the routine has to use PTR #1

```
B1B5:lda &A
```

```
B1B7:pha
```

```
B1B8:lda &B
```

```
B1BA:pha
```

```
B1BB:lda &C
```

```
B1BD:pha
```

Get the offset of PTR #2 into X

```
B1BE:lda &1B
```

```
B1C0:tax
```

Add the offset of PTR #2 to PTR #2, storing the result in A(lsb) and Y(msb)

```
B1C1:clc
```

```
B1C2:adc &19
```

```
B1C4:ldy &1A
```

```
B1C6:bcc &B1CA
```

```
B1C8:iny
```

```
B1C9:clc
```

Subtract two from the value and use it to construct a pointer at (&37). This initially points two characters to the left of PTR #2

```
B1CA:sbc #&1
```

```
B1CC:sta &37
```

```
B1CE:tya
```

```
B1CF:sbc #&0
```

```
B1D1:sta &38
```

Get the name of the procedure or function that is being called

```
B1D3:ldy #&2
```

```
B1D5:jsr &955B
```

If Y is still 2 after the name has been scanned, the name was omitted, so give a 'Bad call' error message

```
B1D8:cpy #&2
```

```
B1DA:beq &B18A
```

Set PTR #2 to point to the end of the name

B1DC:stx &1B

Save the length of the name in &39

B1DE:dey

B1DF:sty &39

Find the catalogue entry for the PROC/FN

B1E1:jsr &945B

If no catalogue entry was found, search memory for the PROC/FN and create a catalogue entry

B1E4:bne &B1E9

B1E6:jmp &B112

Get the address of the PROC/FN from the catalogue and save it in PTR#1

B1E9:ldy #&0

B1EB:lda (&2A),Y

B1ED:sta &B

B1EF:iny

B1F0:lda (&2A),Y

B1F2:sta &C

(Save zero on the machine stack. The number on the top of the stack is used to keep track of the number of parameters)

B1F4:lda #&0

B1F6:pha

B1F7:sta &A

Get the character following the PROC/FN name in the definition

B1F9:jsr &8A97

Search the for parameters if the next character is a '('

B1FC:cmp #&28

B1FE:beq &B24D

Decrement the offset of PTR#1 to make it point to the opening bracket of the defined parameters

B200:dec &A

Save PTR#2 on the machine stack, which keeps track of the address where control must be returned after the PROC/FN exits

B202:lda &1B

B204:pha

B205:lda &19

B207:pha

B208:lda &1A

B20A:pha

Execute the statements comprising the PROC/FN

B20B:jsr &8BA3

Retrieve PTR#2 from the machine stack

B20E:pla

B20F:sta &1A

B211:pla

B212:sta &19

B214:pla**B215:sta &1B**

Retrieve the number of parameters

B217:pla

Skip the next sequence if there were no parameters (or LOCAL variables)

B218:beq &B226

Save the number of remaining parameters to deal with

B21A:sta &3F

Pull the address and type of the parameter

B21C:jsr &BE0B

Replace it with its original value from the BASIC stack

B21F:jsr &8CC1

Continue until all the parameters have been dealt with

B222:dec &3F**B224:bne &B21C**

Retrieve PTR #1 from the machine stack

B226:pla**B227:sta &C****B229:pla****B22A:sta &B****B22C:pla****B22D:sta &A****B22F:pla**

Get the value of the machine stack pointer before the PROC/FN was called

B230:ldy #&0**B232:lda (&4),Y**

Set the machine stack pointer to this address

B234:tax**B235:txs**

Transfer the bytes on the stack

B236:iny**B237:inx****B238:lda (&4),Y****B23A:sta &100,X****B23D:cpx #&FF****B23F:bne &B236**

Increase the BASIC stack pointer to allow for the removal of the machine stack

B241:tya**B242:adc &4****B244:sta &4****B246:bcc &B24A****B248:inc &5**

Get the type of the result of the FN in the accumulator. In the case of procedures, this instruction is not needed

B24A:lda &27

Exit

B24C:rts**Deal with parameters for PROC/FN**

Refer to the section on procedures and functions for an overview of the operation of this routine. As before, a distinction is drawn between the called parameters and the defined parameters

Save the current setting of PTR #2, which points to the calling parameters, because it has to be used to scan the names of the defined parameters

B24D:lda &1B**B24F:pha****B250:lda &19****B252:pha****B253:lda &1A****B255:pha**

Get the name of the next defined parameter

B256:jsr &9582

Give an 'Arguments' error message if the name is invalid

B259:beq &B2B5

Update the offset of PTR #1 to point to the comma or the ')' after the defined parameter name

B25B:lda &1B**B25D:sta &A**

Retrieve the setting of PTR #2

B25F:pla**B260:sta &1A****B262:pla****B263:sta &19****B265:pla****B266:sta &1B**

Retrieve the number of parameters and save it in X

B268:pla**B269:tax**

Save the type and address of the defined parameter on the machine stack

B26A:lda &2C**B26C:pha****B26D:lda &2B****B26F:pha****B270:lda &2A****B272:pha**

Increment the number of parameters and save the result on the machine stack

B273:inx**B274:txa****B275:pha**

Save the value, address and type of the defined parameter variable

B276:jsr &B30D

Get the next character from the defined parameters

B279:jsr &8A97

Deal with the next parameter if the next character is a comma

B27C:cmp #&2C

B27E:beq &B24D

Give an 'Arguments' error if the next character is not the ')' terminating the defined parameters

B280:cmp #&29

B282:bne &B2B5

Save the number of calling parameters as zero

B284:lda #&0

B286:pha

Give an 'Arguments' error message if the first character of the calling parameters area is not an '('

B287:jsr &8A8C

B28A:cmp #&28

B28C:bne &B2B5

Evaluate the parameter

B28E:jsr &9B29

Save its value on the BASIC stack

B291:jsr &BD90

Save its type as a four byte integer on the BASIC stack

B294:lda &27

B296:sta &2D

B298:jsr &BD94

Increment the number of calling parameters

B29B:pla

B29C:tax

B29D:inx

B29E:txa

B29F:pha

Get the next character from the calling parameter area

B2A0:jsr &8A8C

Get the next calling parameter if the next character is a comma

B2A3:cmp #&2C

B2A5:beq &B28E

Give an 'Arguments' error message if the next character is not a ')'

B2A7:cmp #&29

B2A9:bne &B2B5

Get the number of calling parameters. In fact, the number is already in X, so this is a dummy operation

B2AB:pla

Get the number of defined parameters and store the number in &4D and &4E

B2AC:pla**B2AD:sta &4D****B2AF:sta &4E**

Compare the number of calling parameters with the number of defined parameters and if they are the same, proceed to &B2CA, otherwise give an 'Arguments' error message

B2B1:cpx &4D**B2B3:beq &B2CA**

Set up the stack

B2B5:ldx # &FB**B2B7:txs**

Restore the setting of PTR # 1, so that ERL reads correctly

B2B8:pla**B2B9:sta &C****B2BB:pla****B2BC:sta &B**

Give the 'Arguments' error message

B2BE:brk**B2BF:1F-****B2C0:41-A****B2C1:72-r****B2C2:67-g****B2C3:75-u****B2C4:6D-m****B2C5:65-e****B2C6:6E-n****B2C7:74-t****B2C8:73-s****B2C9:00-**

Pull the type of the calling parameter from the BASIC stack

B2CA:jsr &BDEA

Pull the type and address of the defined parameter to the bottom of the IAC. The top byte of the IAC will still contain the type of the calling parameter

B2CD:pla**B2CE:sta &2A****B2D0:pla****B2D1:sta &2B****B2D3:pla****B2D4:sta &2C**

Goto &B2F9 if the defined parameter is a string

B2D6:bmi &B2F9

Give an 'Arguments' error message if the calling parameter is a string

B2D8:lda &2D**B2DA:beq &B2B5**

Save the type of the calling parameter

B2DC:sta &27

Copy the IAC to &37 onwards

B2DE:ldx #&37

B2E0:jsr &BE44

Retrieve the type of the calling parameter

B2E3:lda &27

If it is a real number, branch to &B2F0

B2E5:bpl &B2F0

Discard the floating point number from the top of the BASIC stack, leaving (&4B) pointing to it

B2E7:jsr &BD7E

Load FAC # 1 from (&4B)

B2EA:jsr &A3B5

Skip the code concerned with integers

B2ED:jmp &B2F3

Pull the integer off the BASIC stack

B2F0:jsr &BDEA

Assign the variable

B2F3:jsr &B4B7

Skip the code concerned with strings

B2F6:jmp &B303

Give an 'Arguments' error message if the calling parameter is not a string

B2F9:lda &2D

B2FB:bne &B2B5

Pull the string off the BASIC stack

B2FD:jsr &BDCB

Assign it

B300:jsr &8C21

Repeat the process if there are any parameters left

B303:dec &4D

B305:bne &B2CA

Save the number of parameters on the machine stack

B307:lda &4E

B309:pha

Join the main PROC/FN cde

B30A:jmp &B202

Stack value, address & type of variable

Get the type of the variable

B30D:ldy &2C

Goto &B318 if it is not an integer

B30F:cpy #&4

B311:bne &B318

Copy the integer to &37 onwards

B313:ldx #&37

B315:jsr &BE44

Get the value of the variable

B318:jsr &B32C

Save the status byte, since it contains the type of the variable

B31B:php

Save the value of the variable on the BASIC stack

B31C:jsr &BD90

Retrieve the status register

B31F:plp

Exit if the variable was a string or a real number

B320:beq &B329

B322:bmi &B329

Copy the IAC back down from &37 onwards

B324:ldx #&37

B326:jsr &AF56

Save the type and address of the variable

B329:jmp &BD94

Get the value of a variable

Get the type of the variable

B32C:ldy &2C

Goto &B384 if it is a string

B32E:bmi &B384

Goto &B34F if it is a byte

B330:beq &B34F

Goto &B354 if it is a real number

B332:cpy #&5

B334:beq &B354

Get and store the MSB of the integer

B336:ldy #&3

B338:lda (&2A),Y

B33A:sta &2D

Get and store the next byte

B33C:dey

B33D:lda (&2A),Y

B33F:sta &2C

Save the next byte in X to prevent the address of the value being overwritten

B341:dey

B342:lda (&2A),Y

B344:tax

Get and store the LSB

B345:dey

B346:lda (&2A),Y

B348:sta &2A

Store the byte in X

B34A:stx &2B

Indicate the result is an integer

B34C:lda #&40

Exit

B34E:rts

Copy the single byte into the IAC if it is an integer variable

B34F:lda (&2A),Y

B351:jmp &AEEA

Get and store the LSB of the mantissa of the real number

B354:dey

B355:lda (&2A),Y

B357:sta &34

Get and store the next byte

B359:dey

B35A:lda (&2A),Y

B35C:sta &33

Get and store the next byte

B35E:dey

B35F:lda (&2A),Y

B361:sta &32

Store the MSB of the mantissa in the sign byte

B363:dey

B364:lda (&2A),Y

B366:sta &2E

Get and store the exponent of the number

B368:dey

B369:lda (&2A),Y

B36B:sta &30

Zero the rounding byte and the under/flow byte

B36D:sty &35

B36F:sty &2F

If the mantissa is zero, don't try to insert the true numeric bit

B371:ora &2E

B373:ora &32

B375:ora &33

B377:ora &34

B379:beq &B37F

Get the sign byte and add the true numeric bit

B37B:lda &2E

B37D:ora #&80

Save this as the MSB of the mantissa

B37F:sta &31

Indicate the result is a floating point number

B381:lda #&FF

Exit

B383:rts

If the string is not dynamic (eg \$&700), then goto &B3A7

B384:cpy #&80**B386:beq &B3A7**

Get the length of the string

B388:ldy #&3**B38A:lda (&2A),Y**

Save it

B38C:sta &36

Exit if the string is null

B38E:beq &B3A6

Transfer the address of the string to (&38)

B390:ldy #&1**B392:lda (&2A),Y****B394:sta &38****B396:dey****B397:lda (&2A),Y****B399:sta &37**

Transfer the string to the string buffer

B39B:ldy &36**B39D:dey****B39E:lda (&37),Y****B3A0:sta &600,Y****B3A3:tya****B3A4:bne &B39D**

Exit

B3A6:rts

If the string starts in page zero, then construct the output string by treating the LSB of the address of the string as an ASCII code. There does not appear to be any reason for these instructions

B3A7:lda &2B**B3A9:beq &B3C0**

Transfer the bytes from the string to the string buffer, until a carriage return is encountered

B3AB:ldy #&0**B3AD:lda (&2A),Y****B3AF:sta &600,Y****B3B2:eor #&D****B3B4:beq &B3BA****B3B6:iny****B3B7:bne &B3AD**

Place the length of the string in &36

B3B9:tya
B3BA:sty &36

Exit

B3BC:rts

CHR\$ function routine

Evaluate the ASCII code required

B3BD:jsr &92E3

Construct a string made up of the LSB of the IAC

B3C0:lda &2A

B3C2:jmp &AFC2

Find ERL

This routine works out in which line the most recent error occurred in by examining the contents of PTR # 1. The algorithm is not particularly easy to see, so here is a bastard BASIC version of it:

```

1.ERL = 0
2.BETA = PAGE
3.IF immediate mode THEN RETURN
4.A = ?BETA
5.BETA = BETA + 1
6.IF A <> 13 THEN GOTO 17
7.IF PTR # 1 < BETA THEN RETURN
8.A = ?BETA
9.BETA = BETA + 1
10.IF A > 127 THEN RETURN
11.ERL = A*256
12.A = ?BETA
13.BETA = BETA + 1
14.ERL = ERL + A
15.A = ?BETA
16.BETA = BETA + 1
17.IF PTR # 1 > = BETA THEN GOTO 4
18.RETURN

```

In the above code, BETA represents the pointer constructed by the routine in (&37)

Zero ERL

B3C5:ldy #&0

B3C7:sty &8

B3C9:sty &9

Construct a pointer at (&37) to point to PAGE

B3CB:ldx &18

B3CD:stx &38

B3CF:sty &37

Exit if immediate mode is being used (which will leave ERL at zero)

B3D1:ldx &C

B3D3:cpx #&7

B3D5:beq &B401

Get the LSB of PTR#1 and save it in X

B3D7:ldx &B

Get a character from the pointer at (&37) and increment the pointer

B3D9:jsr &8942

Exit if the character is not a carriage return. This means the line has not yet finished

B3DC:cmp #&D

B3DE:bne &B3F9

Compare the address of the start of the line with PTR#1

B3E0:cpx &37

B3E2:lda &C

B3E4:sbc &38

Exit if PTR#1 is less than the start address of the line

B3E6:bcc &B401

Get a character and increment the pointer

B3E8:jsr &8942

Set the flags according to the character

B3EB:ora #&0

Exit if the byte indicates the end of the program

B3ED:bmi &B401

Save it as the MSB of ERL

B3EF:sta &9

Get the LSB of ERL

B3F1:jsr &8942

B3F4:sta &8

Get the length of the line

B3F6:jsr &8942

Compare the pointer with PTR#1

B3F9:cpx &37

B3FB:lda &C

B3FD:sbc &38

If PTR#1 is greater than the pointer at (&37), then continue the search

B3FF:bcs &B3D9

Exit

B401:rts

BRK handling routine

Find ERL

B402:jsr &B3C5

Turn off TRACE

B405:sty &20

Get ERR

B407:lda (&FD),Y

If ERR is not zero, the error is not fatal, so do not carry out the ON ERROR OFF routine

B409:bne &B413

Set the BASIC error handler to its default

B40B:lda #&33

B40D:sta &16

B40F:lda #&B4

B411:sta &17

Set PTR#1 to the address of the BASIC routine which is set up to handle errors

B413:lda &16

B415:sta &B

B417:lda &17

B419:sta &C

Carry out a RESTORE, clear the BASIC stack and zero the REPEAT, FOR and GOSUB stacks

B41B:jsr &BD3A

Zero the offset of PTR#1

B41E:tax

B41F:stx &A

Clear the VDU queue using OSBYTE &DA. This means that if an error occurred half way through a VDU 23 instruction, the instruction would be ignored by the operating system. Otherwise, the VDU 23 instruction would eat up the first few bytes of the error message

B421:lda #&DA

B423:jsr &FFF4

Acknowledge detection of an ESCAPE condition using OSWORD &7E. Even though most errors are not caused by ESCAPE, this doesn't do any harm

B426:lda #&7E

B428:jsr &FFF4

Set OPT to &FF

B42B:ldx #&FF

B42D:stx &28

Set the machine stack pointer to &FF

B42F:txs

Execute the BASIC statements to deal with the error

B430:jmp &8BA3

BASIC error handler

This section contains a small BASIC program to deal with errors. It reads:

REPORT:IF ERL PRINT " at line ";ERL:END ELSE PRINT:END

(In fact, the ROM version does not include so many spaces)

B433:F6-v

B434:3A-:

B435:E7-g

B436:9E-
B437:F1-q
B438:22-"
B439:20-
B43A:61-a
B43B:74-t
B43C:20-
B43D:6C-l
B43E:69-i
B43F:6E-n
B440:65-e
B441:20-
B442:22-"
B443:3B-;
B444:9E-
B445:3A-:
B446:E0-‘
B447:8B-
B448:F1-q
B449:3A-:
B44A:E0-‘
B44B:0D-

SOUND statement routine

Evaluate the first parameter

B44C:jsr &8821

Do the next part three times for the three remaining parameters

B44F:ldx #&3

Save the parameter on the machine stack

B451:lda &2A

B453:pha

B454:lda &2B

B456:pha

Save the loop counter on the machine stack

B457:txa

B458:pha

Check for a comma and evaluate the next parameter

B459:jsr &92DA

Retrieve the loop counter

B45C:pla

B45D:tax

Continue the process until the counter is zero

B45E:dex

B45F:bne &B451

Check for the end of the statement

B461:jsr &9852

Copy the third parameter to &3D,&3E

B464:lda &2A

B466:sta &3D

B468:lda &2B

B46A:sta &3E

Indicate OSWORD 7 will be used

B46C:ldy #&7

Indicate that there are three two byte parameters left on the stack

B46E:ldx #&5

Join the ENVELOPE code

B470:bne &B48F

ENVELOPE statement routine

Evaluate the first parameter

B472:jsr &8821

Set the loop counter for the remaining 13 parameters

B475:ldx #&D

Save the most recent parameter on the machine stack

B477:lda &2A

B479:pha

Save the loop counter on the machine stack

B47A:txa

B47B:pha

Check the presence of a comma and evaluate the next parameter

B47C:jsr &92DA

Retrieve the loop counter

B47F:pla

B480:tax

Decrement the counter

B481:dex

Continue pushing the parameters until enough have been processed

B482:bne &B477

Check for the end of the statement

B484:jsr &9852

Save the last parameter at &44

B487:lda &2A

B489:sta &44

Indicate that 13 parameters are still on the stack

B48B:ldx #&C

Indicate that OSWORD 8 is required

B48D:ldy #&8

Pull all the remaining parameters from the machine stack

B48F:pla

B490:sta &37,X

B492:dex

B493:bpl &B48F

B495:tya

Make X and Y point to the parameter area

B496:ldx #&37

B498:ldy #&0

Call OSWORD

B49A:jsr &FFF1

Exit

B49D:jmp &8B9B

WIDTH statement routine

Evaluate the width

B4A0:jsr &8821

Check for the end of the statement

B4A3:jsr &9852

Decrement the given WIDTH value

B4A6:ldy &2A

B4A8:dey

Save it as the current terminal width

B4A9:sty &23

Exit

B4AB:jmp &8B9B

Direct jump to the 'Type mismatch' error message

B4AE:jmp &8C0E

Assign to numeric variable

This routine has two entry points: at &B4B1 an expression is evaluated before the assignment takes place, the entry at &B4B4 uses the result of the most recent expression. On entry, the BASIC stack must contain the variable descriptor, comprising the address and type of the variable stored as a four byte integer

Evaluate the expression

B4B1:jsr &9B29

Pull the variable descriptor to &37 onwards

B4B4:jsr &BE0B

If the variable is real, goto &B4E0

B4B7:lda &39

B4B9:cmp #&5

B4BB:beq &B4E0

Give a 'Type mismatch' error if the result of the expression is a string

B4BD:lda &27

B4BF:beq &B4AE

Convert the result to an integer if it is currently expressed as a real number

B4C1:bpl &B4C6

B4C3:jsr &A3E4

Point to the first byte of the integer

B4C6:ldy #&0

Get the LSB of the integer and place it in the variable

B4C8:lda &2A

B4CA:sta (&37),Y

If the variable type is 'byte', exit at this point, since the byte has just been transferred

B4CC:lda &39

B4CE:beq &B4DF

Transfer the remaining bytes

B4D0:lda &2B

B4D2:iny

B4D3:sta (&37),Y

B4D5:lda &2C

B4D7:iny

B4D8:sta (&37),Y

B4DA:lda &2D

B4DC:iny

B4DD:sta (&37),Y

Exit

B4DF:rts

Give a 'Type mismatch' error if the result of the expression is a string

B4E0:lda &27

B4E2:beq &B4AE

Convert it to a floating point number if it is an integer

B4E4:bmi &B4E9

B4E6:jsr &A2BE

Save the exponent of FAC #1

B4E9:ldy #&0

B4EB:lda &30

B4ED:sta (&37),Y

B4EF:iny

Save the MSB of the mantissa of FAC #1

B4F0:lda &2E

B4F2:and #&80

B4F4:sta &2E

B4F6:lda &31

B4F8:and #&7F

B4FA:ora &2E

B4FC:sta (&37),Y

Transfer the remaining bytes of the mantissa

B4FE:iny

B4FF:lda &32

B501:sta (&37),Y

```

B503:iny
B504:lda &33
B506:sta (&37),Y
B508:iny
B509:lda &34
B50B:sta (&37),Y

```

Exit

```

B50D:rts

```

Print a token

This routine prints the character in A, detokenising it if it is greater than 127. COUNT is updated and a carriage return is printed if it is indicated by WIDTH

Save the character in &37

```

B50E:sta &37

```

Output it straight if it is not a token

```

B510:cmp #&80

```

```

B512:bcc &B558

```

Make (&38) point to the start of the token table

```

B514:lda #&71

```

```

B516:sta &38

```

```

B518:lda #&80

```

```

B51A:sta &39

```

Save Y

```

B51C:sty &3A

```

Increment Y from zero until it points to a token

```

B51E:ldy #&0

```

```

B520:iny

```

```

B521:lda (&38),Y

```

```

B523:bpl &B520

```

If the token found is the same as the character being output, goto &8536

```

B525:cmp &37

```

```

B527:beq &B536

```

Point to the start of the next keyword

```

B529:iny

```

Add the displacement to the pointer, and return to &B51E

```

B52A:tya

```

```

B52B:sec

```

```

B52C:adc &38

```

```

B52E:sta &38

```

```

B530:bcc &B51E

```

```

B532:inc &39

```

```

B534:bcs &B51E

```

Point to the first character of the keyword

```

B536:ldy #&0

```


Get the current character

B538:lda (&38),Y

Exit if it is the token marking the end of the keyword

B53A:bmi &B542

Print the character

B53C:jsr &B558

Point to the next character

B53F:iny

Continue the process

B540:bne &B538

Retrieve the contents of Y

B542:ldy &3A

Exit

B544:rts

Print A in hex

Save the byte to be printed

B545:pha

Get the upper nybble of the number into the lower nybble of A

B546:lsr A

B547:lsr A

B548:lsr A

B549:lsr A

Turn it into an ASCII digit and print it

B54A:jsr &B550

Get the original number back

B54D:pla

Isolate the lower nybble

B54E:and #&F

Add seven to the digit if it is over 9

B550:cmp #&A

B552:bcc &B556

B554:adc #&6

Add the ASCII factor

B556:adc #&30

Fall into the print routine

Print the character in A

This routine outputs the character in A, updating COUNT and checking WIDTH

Goto &B567 if the character is not a carriage return

B558:cmp #&D

B55A:bne &B567

Print the carriage return

B55C:jsr &FFEE

Zero COUNT and exit

B55F: jmp &BC28

Print A in hex followed by a space

Print the hex number

B562: jsr &B545

Print a space

B565: lda #&20

Save the character

B567: pha

Compare COUNT with WIDTH

B568: lda &23

B56A: cmp &1E

If COUNT > WIDTH, then move to a new line. If 'WIDTH 0' was given, the WIDTH location will contain 255. COUNT can never be over 255, so a new line is never given

B56C: bcs &B571

B56E: jsr &BC25

Retrieve the character

B571: pla

Increment COUNT

B572: inc &1E

Print the character

B574: jmp (&20E)

Print the LISTO spaces

On entry, A contains the mask to hold against the LISTO byte

Exit if the selected option is not selected

B577: and &1F

B579: beq &B589

Get the level of indentation so far. (This will be the notional figure of &FF for the leading space printed after the line number)

B57B: txa

Exit if the level of indentation is zero

B57C: beq &B589

Print a single space and then exit if the level of indentation is a negative number (specifically, if it is &FF)

B57E: bmi &B565

Print two spaces

B580: jsr &B565

B583: jsr &B558

Continue printing the pairs of spaces until the current indentation level is reached

B586: dex

B587: bne &B580

Exit

B589: rts

LISTO command routine

Increment PTR # 1 past the letter zero

B58A:inc &A

Evaluate the expression

B58C:jsr &9B1D

Check for the end of the statement

B58F:jsr &984C

Ensure the result of the expression is an integer

B592:jsr &92EE

Use the result of the expression as the new LISTO value

B595:lda &2A**B597:sta &1F**

Exit

B599:jmp &8AF6**LIST command routine**

Get the character following the token for 'LIST'

B59C:iny**B59D:lda (&B),Y**

If it is an 'O', execute the code for 'LISTO'

B59F:cmp #&4F**B5A1:beq &B58A**

Zero the indentation count for FOR and REPEAT loops

B5A3:lda #&0**B5A5:sta &3B****B5A7:sta &3C**

Zero the IAC

B5A9:jsr &AED8

Search for a line number after 'LIST'

B5AC:jsr &97DF

Save the flags indicating whether or not it was found

B5AF:php

Save the line number on the stack

B5B0:jsr &BD94

Set the bottom of the IAC to &7FFF, which is the default for the second of the two line numbers

B5B3:lda #&FF**B5B5:sta &2A****B5B7:lda #&7F****B5B9:sta &2B**

Retrieve the flags giving whether or not the first line number was found

B5BB:plp

Skip the next piece of code if the first line was not found

B5BC:bcc &B5CF

Get the next character

B5BE:jsr &8A97

Goto &B5D8 if it is a comma

B5C1:cmp #&2C

B5C3:beq &B5D8

Pull the first line number

B5C5:jsr &BDEA

Push it back again. This makes the stacked line number, which is the first line to be listed, the same as the IAC, which is the last line to be listed. Thus, only one line will be listed

B5C8:jsr &BD94

Decrement the pointer to allow for not finding the comma

B5CB:dec &A

Jump forwards to do the listing

B5CD:bpl &B5DB

Skip the comma if present

B5CF:jsr &8A97

B5D2:cmp #&2C

B5D4:beq &B5D8

B5D6:dec &A

Search for the line number

B5D8:jsr &97DF

Save the number in &31 and &32

B5DB:lda &2A

B5DD:sta &31

B5DF:lda &2B

B5E1:sta &32

Check for the end of the statement

B5E3:jsr &9857

Check for a 'Bad program'

B5E6:jsr &BE6F

Pull the initial line number off the stack

B5E9:jsr &BDEA

Search for the current line number

B5EC:jsr &9970

Save the address of the line in PTR#1

B5EF:lda &3D

B5F1:sta &B

B5F3:lda &3E

B5F5:sta &C

Don't list it if the line does not exist

B5F7:bcc &B60F

Move back one character

B5F9:dey

Branch (always) to &B602

B5FA: bcs &B602

Move to a new line

B5FC: jsr &BC25

Update PTR # 1 and check 'Escape'

B5FF: jsr &986D

Get the line number of the line and save it in the IAC

B602: lda (&B),Y

B604: sta &2B

B606: iny

B607: bcs &B614

B609: sta &2A

Save the offset of the text of the line in &A

B60B: iny

B60C: iny

B60D: sty &A

Subtract the limiting line number from the current line number

B60F: lda &2A

B611: clc

B612: sbc &31

B614: lda &2B

B616: sbc &32

If the current line number is less than the limiting line number, exit

B618: bcc &B61D

B61A: jmp &8AF6

Print the line number

B61D: jsr &9923

Set the 'quotes mode' flag to &FF. If this flag contains zero, the text being listed is between quote symbols, so tokens must not be expanded

B620: ldx #&FF

B622: stx &4D

Keeping a mock indentation level of &FF, print the leading space before the text of the line

B624: lda #&1

B626: jsr &B577

Print the FOR indentation spaces

B629: ldx &3B

B62B: lda #&2

B62D: jsr &B577

Print the REPEAT indentation spaces

B630: ldx &3C

B632: lda #&4

B634: jsr &B577

Get the pointer to the first character of the text of the line

B637: ldy &A

Get the current character

B639:lda (&B),Y

Exit if it is a carriage return

B63B:cmp #&D**B63D:beq &B5FC**

If the character is a quote symbol, flip the quote status flag and print the symbol

B63F:cmp #&22**B641:bne &B651****B643:lda #&FF****B645:eor &4D****B647:sta &4D**

Print the quote symbol

B649:lda #&22**B64B:jsr &B558**

Point to the next character

B64E:iny

Go back and get a new character

B64F:bne &B639

If quotes mode is in effect, simply print the character and go back for a new one

B651:bit &4D**B653:bpl &B64B**

If the character is not a line number token, skip the next bit of code

B655:cmp #&8D**B657:bne &B668**

Decode the line number

B659:jsr &97EB

Save the offset of PTR #1

B65C:sty &A

Set the field width to zero

B65E:lda #&0**B660:sta &14**

Print the line number

B662:jsr &991F

Get a new character

B665:jmp &B637

If the character is the token for 'FOR', increment the relevant indentation count

B668:cmp #&E3**B66A:bne &B66E****B66C:inc &3B**

Skip if the character is not the token for 'NEXT'

B66E:cmp #&ED**B670:bne &B678**

Get the current 'FOR' indentation count

B672:ldx &3B

Exit if it is zero

B674:beq &B678

Decrement it

B676:dec &3B

Increment the 'REPEAT' indentation count if the character is the token for 'REPEAT'

B678:cmp #&F5

B67A:bne &B67E

B67C:inc &3C

Decrement the 'REPEAT' indentation count if the character is the token for 'UNTIL' and the indentation count is not zero

B67E:cmp #&FD

B680:bne &B688

B682:ldx &3C

B684:beq &B688

B686:dec &3C

Print the character

B688:jsr &B50E

Point to the next character

B68B:iny

Go back and get the next character

B68C:bne &B639

'No FOR' error message

B68E:brk

B68F:20-

B690:4E-N

B691:6F-o

B692:20-

B693:E3-c

B694:00-

NEXT statement routine

Get the name of the variable

B695:jsr &95C9

Jump to &B6A3 if the variable is valid and defined

B698:bne &B6A3

Get the current FOR stack pointer

B69A:ldx &26

Give a 'No FOR' error if the stack is empty

B69C:beq &B68E

Goto &B6D7 if the variable name is invalid

B69E:bcs &B6D7

Give a 'Syntax error'

B6A0:jmp &982A

Give a 'Syntax error' if the variable is a string

B6A3: bcs &B6A0

Get the FOR stack pointer

B6A5: ldx &26

Give a 'No FOR' error message if the stack is empty

B6A7: beq &B68E

Compare the variable information block obtained from the variable name with the variable information block stored on the FOR stack

B6A9: lda &2A

B6AB: cmp &4F1,X

B6AE: bne &B6BE

B6B0: lda &2B

B6B2: cmp &4F2,X

B6B5: bne &B6BE

B6B7: lda &2C

B6B9: cmp &4F3,X

Goto &B6D7 if they are the same

B6BC: beq &B6D7

Unstack the top record on the FOR stack

B6BE: txa

B6BF: sec

B6C0: sbc #&F

B6C2: tax

B6C3: stx &26

If the stack is not empty, try the next information block on the stack

B6C5: bne &B6A9

Give a 'Can't Match FOR' error message

B6C7: brk

B6C8: 21-!

B6C9: 43-C

B6CA: 61-a

B6CB: 6E-n

B6CC: 27-'

B6CD: 74-t

B6CE: 20-

B6CF: 4D-M

B6D0: 61-a

B6D1: 74-t

B6D2: 63-c

B6D3: 68-h

B6D4: 20-

B6D5: E3-c

B6D6: 00-

Retrieve the address of the variable

B6D7: lda &4F1,X

B6DA:sta &2A

B6DC:lda &4F2,X

B6DF:sta &2B

Retrieve the type of the variable

B6E1:ldy &4F3,X

Goto &B766 if the variable is real

B6E4:cpy #&5

B6E6:beq &B766

Get the LSB of the current value of the loop variable

B6E8:ldy #&0

B6EA:lda (&2A),Y

Add the step size

B6EC:adc &4F4,X

Save the result

B6EF:sta (&2A),Y

Save the result in &37 as well

B6F1:sta &37

Add the second bytes, storing the result in &38 as well

B6F3:iny

B6F4:lda (&2A),Y

B6F6:adc &4F5,X

B6F9:sta (&2A),Y

B6FB:sta &38

Add the third bytes, storing the result in &39 as well

B6FD:iny

B6FE:lda (&2A),Y

B700:adc &4F6,X

B703:sta (&2A),Y

B705:sta &39

Add the fourth bytes, storing the result in Y as well

B707:iny

B708:lda (&2A),Y

B70A:adc &4F7,X

B70D:sta (&2A),Y

B70F:tay

Subtract the terminating value of the loop from the new value of the loop variable

B710:lda &37

B712:sec

B713:sbc &4F9,X

B716:sta &37

B718:lda &38

B71A:sbc &4FA,X

B71D:sta &38

B71F:lda &39

B721:sbc &4FB,X

B724:sta &39

B726:tya

B727:sbc &4FC,X

If the result is zero, goto &B741

B72A:ora &37

B72C:ora &38

B72E:ora &39

B730:beq &B741

Exclusive-or the MSB of the new loop variable value with the LSBs of the step size and the terminating value. If the accumulator is positive on exit, two of the quantities must have been negative and one positive, or all three must have been positive. This implies that the loop should terminate when the new loop variable value is greater than the terminating value. If the result is negative, the loop should terminate when the new loop variable value is less than the terminating value.

B732:tya

B733:eor &4F7,X

B736:eor &4FC,X

Goto &B73F if this is a 'upwards' loop

B739:bpl &B73F

Carry out another iteration if the new value of the variable is greater than the limit

B73B:bcs &B741

Otherwise, terminate the loop

B73D:bcc &B751

Terminate the loop if the loop variable is greater than the limit

B73F:bcs &B751

Set PTR #1 to the address of the start of the loop

B741:ldy &4FE,X

B744:lda &4FF,X

B747:sty &B

B749:sta &C

Check the 'Escape' flag

B74B:jsr &9877

Execute the body of the loop

B74E:jmp &8BA3

Subtract 15 from the FOR stack pointer to remove the top record

B751:lda &26

B753:sec

B754:sbc #&F

B756:sta &26

Update PTR #1 from PTR #2

B758:ldy &1B

B75A:sty &A

Get the next character

B75C:jsr &8A97

Exit if it is not a comma

B75F:cmp #&2C

B761:bne &B7A1

Go back to deal with the next variable name

B763:jmp &B695

Get the current value of loop variable into FAC#1

B766:jsr &B354

Construct a pointer at (&4B) that points to the record in the top FOR stack entry

B769:lda &26

B76B:clc

B76C:adc #&F4

B76E:sta &4B

B770:lda #&5

B772:sta &4C

Add the step size to the loop variable

B774:jsr &A500

Make (&37) point to the value of the variable

B777:lda &2A

B779:sta &37

B77B:lda &2B

B77D:sta &38

Move FAC#1 to the variable

B77F:jsr &B4E9

Save the FOR stack pointer in &27

B782:lda &26

B784:sta &27

Make (&4B) point to the terminating value of the top record on the FOR stack

B786:clc

B787:adc #&F9

B789:sta &4B

B78B:lda #&5

B78D:sta &4C

Set FAC#1 = FAC#1-terminating value

B78F:jsr &9A5F

End the loop if the result is zero

B792:beq &B741

If the step size is negative, goto &B79D

B794:lda &4F5,X

B797:bmi &B79D

Carry on with the loop if the loop variable is less than the terminating value

B799:bcsl &B741

Otherwise, terminate the loop

B79B:bcc &B751

Carry on with the loop if the loop variable is greater than the terminating value

B79D:bcc &B741

B79D:bcc &B741

Otherwise, terminate the loop

B79F:bcs &B751

Exit

B7A1:jmp &8B96

'FOR variable' error message

B7A4:brk

B7A5:22-"

B7A6:E3-c

B7A7:20-

B7A8:76-v

B7A9:61-a

B7AA:72-r

B7AB:69-i

B7AC:61-a

B7AD:62-b

B7AE:6C-l

B7AF:65-e

'Too many FORs' error message

B7B0:brk

B7B1:23-#

B7B2:54-T

B7B3:6F-o

B7B4:6F-o

B7B5:20-

B7B6:6D-m

B7B7:61-a

B7B8:6E-n

B7B9:79-y

B7BA:20-

B7BB:E3-c

B7BC:73-s

'No TO' error message

B7BD:brk

B7BE:24-\$

B7BF:4E-N

B7C0:6F-o

B7C1:20-

B7C2:B8-8

B7C3:00-

FOR statement routine

Get the name of the loop variable

B7C4:jsr &9582

Give a 'FOR variable' error message if the variable name is invalid or a string

B7C7:beq &B7A4

B7C9:bcs &B7A4

Save the type and address of the variable

B7CB:jsr &BD94

Check for an equals sign

B7CE:jsr &9841

Evaluate the expression giving the starting value of the loop and assign this value to the variable

B7D1:jsr &B4B1

Give a 'Too many FORs' error message if more than 10 FOR loops are already set up

B7D4:ldy &26

B7D6:cpy #&96

B7D8:bcs &B7B0

Save the type and address of the variable on the FOR stack

B7DA:lda &37

B7DC:sta &500,Y

B7DF:lda &38

B7E1:sta &501,Y

B7E4:lda &39

B7E6:sta &502,Y

Save the type of the variable in X

B7E9:tax

Give a 'No TO' error message if the next character is not the token for 'TO'

B7EA:jsr &8A8C

B7ED:cmp #&B8

B7EF:bne &B7BD

Goto &B84F if this is a real FOR loop

B7F1:cpx #&5

B7F3:beq &B84F

Evaluate the upper limit of the loop, and ensure the result is an integer

B7F5:jsr &92DD

Get the current index into the FOR stack

B7F8:ldy &26

Save the terminating value of the loop on the FOR stack

B7FA:lda &2A

B7FC:sta &508,Y

B7FF:lda &2B

B801:sta &509,Y

B804:lda &2C

B806:sta &50A,Y

B809:lda &2D

B80B:sta &50B,Y

Set the IAC to 1

B80E:lda #&1

B810:jsr &AED8

Get the next character

B813:jsr &8A8C

If it is not the token for 'STEP', skip evaluating the STEP size

B816:cmp #&88**B818:bne &B81F**

Evaluate the STEP size, ensuring the result is an integer

B81A:jsr &92DD

Get the offset from PTR # 2

B81D:ldy &1B

Update PTR # 2

B81F:sty &A

Save the STEP value on the FOR stack

B821:ldy &26**B823:lda &2A****B825:sta &503,Y****B828:lda &2B****B82A:sta &504,Y****B82D:lda &2C****B82F:sta &505,Y****B832:lda &2D****B834:sta &506,Y**

Check for the end of the statement, and move PTR # 1 to the start of the next statement

B837:jsr &9880

Save PTR # 1 on the FOR stack to indicate where control should return to at the end of the loop

B83A:ldy &26**B83C:lda &B****B83E:sta &50D,Y****B841:lda &C****B843:sta &50E,Y**

Add 15 to the FOR stack pointer

B846:clc**B847:tya****B848:adc #&F****B84A:sta &26**

Exit

B84C:jmp &8BA3

Evaluate the terminating value for the loop

B84F:jsr &9B29

Ensure it is real

B852:jsr &92FD

Make (&4B) point to the terminating value area of the FOR stack

B855:lda &26**B857:clc**

B858:adc #&8**B85A:sta &4B****B85C:lda #&5****B85E:sta &4C**

Copy the terminating value to the FOR stack

B860:jsr &A38D

Set FAC#1 to 1

B863:jsr &A699

Skip the next few bytes if the next character is not the token for 'STEP'

B866:jsr &8A8C**B869:cmp #&88****B86B:bne &B875**

Evaluate the STEP size

B86D:jsr &9B29

Ensure it is real

B870:jsr &92FD

Get the offset of PTR#2

B873:ldy &1B

Update the offset of PTR#1

B875:sty &A

Set (&4B) to point to the step size area of the FOR stack

B877:lda &26**B879:clc****B87A:adc #&3****B87C:sta &4B****B87E:lda #&5****B880:sta &4C**

Save the STEP size on the FOR stack

B882:jsr &A38D

Join the end of the integer code

B885:jmp &B837**GOSUB statement routine**

Look for a line number after the token for 'GOSUB', and find it in the program

B888:jsr &B99A

Check for the end of the statement

B88B:jsr &9857

If the GOSUB stack pointer is greater than 26, give a 'Too many GOSUBs' error message

B88E:ldy &25**B890:cpy #&1A****B892:bcs &B8A2**

Save the return address on the GOSUB stack

B894:lda &B**B896:sta &5CC,Y**

B899:lda &C

B89B:sta &5E6,Y

Increment the GOSUB stack pointer

B89E:inc &25

Join the code at the end of the GOTO routine

B8A0:bcc &B8D2

'Too many GOSUBs' error message

B8A2:brk

B8A3:25-%

B8A4:54-T

B8A5:6F-o

B8A6:6F-o

B8A7:20-

B8A8:6D-m

B8A9:61-a

B8AA:6E-n

B8AB:79-y

B8AC:20-

B8AD:E4-d

B8AE:73-s

'No GOSUB' error message

B8AF:brk

B8B0:26-&

B8B1:4E-N

B8B2:6F-o

B8B3:20-

B8B4:E4-d

B8B5:00-

RETURN statement routine

Check for the end of the statement

B8B6:jsr &9857

Give a 'No GOSUB' error message if the GOSUB stack is empty

B8B9:ldx &25

B8BB:beq &B8AF

Remove the top entry from the GOSUB stack

B8BD:dec &25

Get the address after the corresponding GOSUB

B8BF:ldy &5CB,X

B8C2:lda &5E5,X

Set PTR#1 to that address

B8C5:sty &B

B8C7:sta &C

Exit

B8C9:jmp &8B9B

GOTO statement routine

Get a line number after the token for 'GOTO', and find its address

B8CC:jsr &B99A

Check for the end of the statement

B8CF:jsr &9857

Give a 'TRACE' output by printing the new line number if required

B8D2:lda &20

B8D4:beq &B8D9

B8D6:jsr &9905

Set PTR # 1 to the address of the new line

B8D9:ldy &3D

B8DB:lda &3E

B8DD:sty &B

B8DF:sta &C

Exit

B8E1:jmp &8BA3

ON ERROR OFF statement routine

Check for the end of the statement

B8E4:jsr &9857

Set the error handler address to the default ROM routine

B8E7:lda #&33

B8E9:sta &16

B8EB:lda #&B4

B8ED:sta &17

Exit

B8EF:jmp &8B9B

ON ERROR statement routine

Get the next character

B8F2:jsr &8A97

If it is the token for 'OFF', goto &B8E4 to execute the code for 'ON ERROR OFF'

B8F5:cmp #&87

B8F7:beq &B8E4

Adjust PTR # 1 to allow for not finding the 'OFF' token

B8F9:ldy &A

B8FB:dey

B8FC:jsr &986D

Set the error handler address to the current address indicated by PTR # 1

B8FF:lda &B

B901:sta &16

B903:lda &C

B905:sta &17

Exit

B907:jmp &8B7D

'ON syntax' error message

B90A:brk
 B90B:27-'
 B90C:EE-n
 B90D:20-
 B90E:73-s
 B90F:79-y
 B910:6E-n
 B911:74-t
 B912:61-a
 B913:78-x
 B914:00-

ON statement routine

Goto the 'ON ERROR ...' routine if the next character is the token for 'ERROR'

B915:jsr &8A97
 B918:cmp #&85
 B91A:beq &B8F2

Decrement the offset of PTR # 1 to allow for not finding the 'ERROR' token

B91C:dec &A

Evaluate the integer expression given in the statement

B91E:jsr &9B1D
 B921:jsr &92F0

Make PTR # 1 point to the character after the 'GOTO' or 'GOSUB'

B924:ldy &1B
 B926:iny
 B927:sty &A

Give an 'ON syntax' error message if the next character is not the token for 'GOTO' or 'GOSUB'

B929:cpx #&E5
 B92B:beq &B931
 B92D:cpx #&E4
 B92F:bne &B90A

Save the token on the machine stack

B931:txa
 B932:pha

Assume no match will be found if the IAC is greater than 255

B933:lda &2B
 B935:ora &2C
 B937:ora &2D
 B939:bne &B97D

Assume no match will be found if the IAC is zero

B93B:ldx &2A
 B93D:beq &B97D

Decrement the ON count

B93F:dex

If it becomes zero, the line number pointed to by PTR # 1 is the one to jump to, so goto &B95C

B940:beq &B95C

Get the next character

B942:ldy &A**B944:lda (&B),Y****B946:iny**

Abort the search if it is a carriage return, a colon or the token for 'ELSE'

B947:cmp #&D**B949:beq &B97D****B94B:cmp #&3A****B94D:beq &B97D****B94F:cmp #&8B****B951:beq &B97D**

Get a new character if it is not a comma

B953:cmp #&2C**B955:bne &B944**

Continue the search if the current line number is not the one corresponding to the IAC

B957:dex**B958:bne &B944**

Save the offset of PTR # 1

B95A:sty &A

Decode the line number

B95C:jsr &B99A

Retrieve the token for 'GOTO' or 'GOSUB'

B95F:pla

Goto &B96A if it is the token for 'GOSUB'

B960:cmp #&E4**B962:beq &B96A**

Check the 'Escape' flag

B964:jsr &9877

Join the GOTO routine

B967:jmp &B8D2

Find the colon or carriage return at the end of the statement

B96A:ldy &A**B96C:lda (&B),Y****B96E:iny****B96F:cmp #&D****B971:beq &B977****B973:cmp #&3A****B975:bne &B96C**

Set PTR # 1 to this address

B977:dey**B978:sty &A**

Join the GOSUB routine

B97A: jmp &B88B

Get the next character

B97D: ldy &A

(Remove the token from the stack, since it is no longer needed)

B97F: pla

B980: lda (&B),Y

B982: iny

Goto &B995 if the character is the token for 'ELSE'

B983: cmp #&8B

B985: beq &B995

Get a new character if it is not the carriage return marking the end of the line

B987: cmp #&D

B989: bne &B980

Give an 'ON range' error message when the end of the line is reached

B98B: brk

B98C: 28-(

B98D: EE-n

B98E: 20-

B98F: 72-r

B990: 61-a

B991: 6E-n

B992: 67-g

B993: 65-e

B994: 00-

Save the offset of the 'ELSE'

B995: sty &A

Join the ELSE code in the IF routine

B997: jmp &98E3

Retrieve line number and locate it

This routine searches for a line number at PTR # 1, and then returns the address of the line in (&3D)

Decode the line number at PTR # 1

B99A: jsr &97DF

Jump forwards if a line number was found

B99D: bcs &B9AF

Evaluate the expression giving the line number

B99F: jsr &9B1D

Ensure the result is an integer

B9A2: jsr &92F0

Update PTR # 1

B9A5: lda &1B

B9A7: sta &A

Make sure the bottom two bytes of the line number are less than 32768

B9A9:lda &2B

B9AB:and #&7F

B9AD:sta &2B

Search the program for the line

B9AF:jsr &9970

Give a 'No such line' error message if it is not found

B9B2:bec &B9B5

Exit

B9B4:rts

'No such line' error message

B9B5:brk

B9B6:29-)

B9B7:4E-N

B9B8:6F-o

B9B9:20-

B9BA:73-s

B9BB:75-u

B9BC:63-c

B9BD:68-h

B9BE:20-

B9BF:6C-l

B9C0:69-i

B9C1:6E-n

B9C2:65-e

B9C3:00-

Direct jump to the 'Type mismatch' error message

B9C4:jmp &8C0E

Direct jump to the 'Syntax error' message

B9C7:jmp &982A

Update PTR #1

B9CA:sty &A

Exit

B9CC:jmp &8B98

INPUT # statement routine

Decrement PTR #1 past the hash character

B9CF:dec &A

Decode the handle of the file

B9D1:jsr &BFA9

Update PTR #1 from PTR #2

B9D4:lda &1B

B9D6:sta &A

Save the handle in &4D

B9D8:sty &4D

Get the next character

B9DA:jsr &8A97

B9DD:cmp #&2C

Exit if it is not a comma

B9DF:bne &B9CA

Save the file handle on the machine stack

B9E1:lda &4D

B9E3:pha

Get the variable name

B9E4:jsr &9582

Give a 'Syntax error' if it is invalid

B9E7:beq &B9C7

Update PTR#1 from PTR#2

B9E9:lda &1B

B9EB:sta &A

Retrieve the file handle

B9ED:pla

Save the file handle

B9EE:sta &4D

Save the status of the variable name

B9F0:php

Save the type and address of the variable

B9F1:jsr &BD94

Get the type of the variable on the media

B9F4:ldy &4D

B9F6:jsr &FFD7

Save the type

B9F9:sta &27

Retrieve the type of the variable name

B9FB:plp

Goto &BA19 if it is numeric

B9FC:bcc &BA19

Give a 'Type mismatch' error if the quantity on the filing system is not a string

B9FE:lda &27

BA00:bne &B9C4

Get the length of the string into the string length location and X

BA02:jsr &FFD7

BA05:sta &36

BA07:tax

Exit if the string is null

BA08:beq &BA13

Get the next character of the string

BA0A:jsr &FFD7

Save it in the buffer

BA0D:sta &5FF,X

Decrement the number of characters left on the media

BA10:dex

If there are some left, go back to store them

BA11:bne &BA0A

Assign the string

BA13:jsr &8C1E

Go back for the next variable

BA16:jmp &B9DA

Give a 'Type mismatch' error if the thing on the media is a string

BA19:lda &27**BA1B:beq &B9C4**

Goto &BA2B if the thing on the media is a real number

BA1D:bmi &BA2B

Get the integer from the media to the IAC

BA1F:ldx #&3**BA21:jsr &FFD7****BA24:sta &2A,X****BA26:dex****BA27:bpl &BA21**

Join the floating point code

BA29:bmi &BA39

Get the real number from the media to the buffer at &46C onwards

BA2B:ldx #&4**BA2D:jsr &FFD7****BA30:sta &46C,X****BA33:dex****BA34:bpl &BA2D**

Unpack the number to FAC#1

BA36:jsr &A3B2

Assign the variable

BA39:jsr &B4B4

Go back for the next variable name

BA3C:jmp &B9DA

Restore the stack

BA3F:pla**BA40:pla**

Exit

BA41:jmp &8B98**INPUT statement routine**

Several flags are used by this routine. If bit 7 of &4D is set, the system precedes getting a line of input by printing a '?'. If bit 6 is set, the LINE option is currently being used. Location &4E contains the current offset into the input buffer. This will be &FF if the system expects the user to enter a new line of text for an item, otherwise it will point to

the text which will be used next. The process becomes clearer when the routine is studied

Get the next character

BA44:jsr &8A97

Join the INPUT # code if it is a hash character

BA47:cmp #&23

BA49:beq &B9CF

Goto &BA52 if it is the token for 'LINE'

BA4B:cmp #&86

BA4D:beq &BA52

Decrement PTR # 1 past the character if it was not 'LINE' or '#'

BA4F:dec &A

Clear the carry flag

BA51:clc

Place the carry flag in the top bit of &4D. Thus, if LINE is used, the top bit will be set, otherwise it will be clear

BA52:ror &4D

Move the bit into bit 6. It can now be tested via the overflow flag

BA54:lsr &4D

Set the flag at &4E to &FF.

BA56:lda #&FF

BA58:sta &4E

Execute any INPUT items present

BA5A:jsr &8E8A

Goto &BA69 if no item was found

BA5D:bcs &BA69

Execute any more items

BA5F:jsr &8E8A

Check for another item if one was just found

BA62:bcc &BA5F

Set the flag at &4E back to &FF. This means that you cannot type '1,2' in response to 'INPUT "ENTER FIRST" A "ENTER SECOND" B'

BA64:ldx #&FF

BA66:stx &4E

Clear the carry flag to indicate that an item was found, so the '?' prompt is not required

BA68:clc

Save the prompt status

BA69:php

Discard the current setting of bit 7 of the &4D flag location

BA6A:asl &4D

Use the carry flag to indicate the prompt status

BA6C:plp

BA6D:ror &4D

Repeat the process if the next character is a comma or a semi-colon

BA6F:cmp #&2C

BA71:beq &BA5A**BA73:cmp #&3B****BA75:beq &BA5A**

Decrement PTR # 1 to allow for not finding a comma or a semi-colon

BA77:dec &A

Save the flags on the machine stack. Otherwise, 'INPUT ARRAY(FNa)', where FNa contains another 'INPUT' statement, would have disastrous effects

BA79:lda &4D**BA7B:pha****BA7C:lda &4E****BA7E:pha**

Get the name of the variable

BA7F:jsr &9582

Exit if the variable is invalid

BA82:beq &BA3F

Retrieve the flags

BA84:pla**BA85:sta &4E****BA87:pla****BA88:sta &4D**

Update PTR # 1 from PTR # 2

BA8A:lda &1B**BA8C:sta &A**

Save the status of the variable name

BA8E:php

Goto &BA99 if the 'LINE' option is in use. This prevents a test being made of the flag at &4E, since the use of 'LINE' indicates that a new line of text must be entered for each variable being input

BA8F:bit &4D**BA91:bvs &BA99**

If there are some unscanned characters left over from the last variable, don't bother to get a new line of text

BA93:lda &4E**BA95:cmp #&FF****BA97:bne &BAB0**

Give the '?' prompt if required

BA99:bit &4D**BA9B:bpl &BAA2****BA9D:lda #&3F****BA9F:jsr &B558**

Input a string to the string buffer

BAA2:jsr &BBFC

Save the length of the string

BAA5:sty &36

Indicate that a prompt will not be required for the next variable to be input

BAA7:asl &4D

BAA9:clc

BAAA:ror &4D

Jump to &4D if 'LINE' is being used

BAAC:bit &4D

BAAE:bvs &BACD

Save the offset of PTR # 2. This will be zero if a new string was entered, or the contents of &4E if the old one is being used

BAB0:sta &1B

Set the base of PTR # 2 to point to the string buffer

BAB2:lda #&0

BAB4:sta &19

BAB6:lda #&6

BAB8:sta &1A

Move the part of the string to be used down so that it starts at &600, with its length in &36

BABA:jsr &ADAD

Get the next character

BABD:jsr &8A8C

Jump forwards if it is a comma

BAC0:cmp #&2C

BAC2:beq &BACA

Skip any other characters until a carriage return is encountered

BAC4:cmp #&D

BAC6:bne &BABD

If a comma was not encountered, prime Y so that it will be stored in &4E as &FF

BAC8:ldy #&FE

Increment Y

BACA:iny

Save it as the offset of the next character

BACB:sty &4E

Retrieve the status of the variable name

BACD:plp

Goto &BADC if it was a string

BACE:bcs &BADC

Save the type and address of the variable

BAD0:jsr &BD94

Convert the string to a number

BAD3:jsr &AC34

Assign the number to the variable

BAD6:jsr &B4B4

Go back for the next item

BAD9:jmp &BA5A

Set the type to string

BADC:lda #&0

BADE:sta &27

Assign the string to the variable

BAE0:jsr &8C21

Return for the next item

BAE3:jmp &BA5A

RESTORE statement routine

Set (&3D) to PAGE

BAE6:ldy #&0

BAE8:sty &3D

BAEA:ldy &18

BAEC:sty &3E

Get the next character

BAEE:jsr &8A97

Ignore it by decrementing PTR # 1

BAF1:dec &A

Skip searching for a line number if the next character is a colon, a carriage return or the token for 'ELSE'

BAF3:cmp #&3A

BAF5:beq &BB07

BAF7:cmp #&D

BAF9:beq &BB07

BAFB:cmp #&8B

BAFD:beq &BB07

Search for a line number - the address of the line will be in (&3D)

BAFF:jsr &B99A

Add one to (&3D)

BB02:ldy #&1

BB04:jsr &BE55

Check for the end of the statement

BB07:jsr &9857

Set the DATA pointer to the address of the line

BB0A:lda &3D

BB0C:sta &1C

BB0E:lda &3E

BB10:sta &1D

Exit

BB12:jmp &8B9B

Get the next character

BB15:jsr &8A97

READ the next item if it is a comma

BB18:cmp #&2C

BB1A:beq &BB1F

Exit

BB1C:jmp &8B96

READ statement routine

Read the address and type of the variable name

BB1F:jsr &9582

If the variable is invalid, go back to check for a comma

BB22:beq &BB15

Goto &BB32 if the variable is a string variable

BB24:bcs &BB32

Point to the next DATA item

BB26:jsr &BB50

Save the address and type of the variable

BB29:jsr &BD94

Assign the value to the variable

BB2C:jsr &B4B1

Join the end of the string item code

BB2F:jmp &BB40

Point to the next DATA item

BB32:jsr &BB50

Save the address and type of the variable on the stack

BB35:jsr &BD94

Read the string into the string buffer

BB38:jsr &ADAD

Set the type to zero, indicating a string is in the string buffer

BB3B:sta &27

Assign the string to the variable

BB3D:jsr &8C1E

Update the DATA pointer from the final contents of PTR#2

BB40:clc

BB41:lda &1B

BB43:adc &19

BB45:sta &1C

BB47:lda &1A

BB49:adc #&0

BB4B:sta &1D

Go back and search for another variable name

BB4D:jmp &BB15

Point to the next DATA item

On exit, PTR#2 points to the comma or 'DATA' token before the next DATA item

Update PTR#1 from PTR#2

BB50:lda &1B

BB52:sta &A

Set PTR#2 to the current setting of the DATA pointer

BB54:lda &1C

BB56:sta &19

BB58:lda &1D

BB5A:sta &1A

BB5C:ldy #&0

BB5E:sty &1B

Get the next character from PTR#2

BB60:jsr &8A8C

Exit if it is a comma or the token for 'DATA'

BB63:cmp #&2C

BB65:beq &BBB0

BB67:cmp #&DC

BB69:beq &BBB0

Jump forwards if it is the carriage return at the end of the line

BB6B:cmp #&D

BB6D:beq &BB7A

Get the next character

BB6F:jsr &8A8C

Exit if it is a comma

BB72:cmp #&2C

BB74:beq &BBB0

Repeat the process until a carriage return is encountered

BB76:cmp #&D

BB78:bne &BB6F

Get the LSB of the line number of the next line

BB7A:ldy &1B

BB7C:lda (&19),Y

Give an 'Out of DATA' error message if the end of the program has been reached

BB7E:bmi &BB9C

Get the length of the line into X

BB80:iny

BB81:iny

BB82:lda (&19),Y

BB84:tax

Get the next character of the text of the line

BB85:iny

BB86:lda (&19),Y

Skip any spaces

BB88:cmp #&20

BB8A:beq &BB85

If it is the word 'DATA', exit

BB8C:cmp #&DC

BB8E:beq &BBAD

Add the length of the line to the pointer and return to continue the search

BB90:txa

BB91:clc

BB92:adc &19

BB94:sta &19
BB96:bcc &BB7A
BB98:inc &1A
BB9A:bcs &BB7A

'Out of DATA' error message

BB9C:brk
BB9D:2A-*
BB9E:4F-O
BB9F:75-u
BBA0:74-t
BBA1:20-
BBA2:6F-o
BBA3:66-f
BBA4:20-
BBA5:DC-/

'No REPEAT' error message

BBA6:brk
BBA7:2B- +
BBA8:4E-N
BBA9:6F-o
BBAA:20-
BBAB:F5-u
BBAC:00-

Increment the pointer to the token for the word 'DATA'

BBAD:iny

Update the RAM image pointer

BBAE:sty &1B

Exit

BBB0:rts

UNTIL statement routine

Evaluate the condition for the loop ending

BBB1:jsr &9B1D

Check for the end of the statement

BBB4:jsr &984C

Ensure the result of the condition is an integer

BBB7:jsr &92EE

Get the current REPEAT stack pointer

BBBA:ldx &24

Give a 'No REPEAT' error message if the stack is empty

BBBC:beq &BBA6

Check whether the IAC is zero

BBBE:lda &2A

BBC0:ora &2B

BBC2:ora &2C**BBC4:ora &2D**

Jump forwards if it does

BBC6:beq &BBCD

Terminate the loop by removing the top element from the REPEAT stack

BBC8:dec &24

Exit

BBCA:jmp &8B9B

Get the address of the character after REPEAT

BBCD:ldy &5A3,X**BBD0:lda &5B7,X**

Join the end of the GOTO code to pass control to the address

BBD3:jmp &B8DD

'Too many REPEATs' error message

BBD6:brk**BBD7:2C-,****BBD8:54-T****BBD9:6F-o****BBDA:6F-o****BBDB:20-****BBDC:6D-m****BBDD:61-a****BBDE:6E-n****BBDF:79-y****BBE0:20-****BBE1:F5-u****BBE2:73-s****BBE3:00-****REPEAT statement routine**

Get the current REPEAT stack pointer

BBE4:ldx &24

Give a 'Too many REPEATs' error message if the stack contains 20 or more entries

BBE6:cpx #&14**BBE8:bcx &BBD6**

Update PTR # 1 to point to the character after the token for 'REPEAT' and check the 'Escape' flag

BBEA:jsr &986D

Save the address on the REPEAT stack

BBED:lda &B**BBEF:sta &5A4,X****BBF2:lda &C****BBF4:sta &5B8,X**

Increment the REPEAT stack pointer

BBF7:inc &24

Exit

BBF9:jmp &8BA3**Enter a string to string buffer**

Place buffer address in Y and A

BBFC:ldy #&0**BBFE:lda #&6**

Join the main string input routine

BC00:bne &BC09**Enter a string to keyboard buffer**

This routine should be entered with the character to be used as a prompt in A

Output the prompt

BC02:jsr &B558

Place the keyboard buffer address in Y and A

BC05:ldy #&0**BC07:lda #&7**

Save the buffer address in the OSWORD parameter area

BC09:sty &37**BC0B:sta &38**

Indicate a maximum length of 238 characters

BC0D:lda #&EE**BC0F:sta &39**

Indicate that the space character is the lowest acceptable code

BC11:lda #&20**BC13:sta &3A**

Indicate that &FF is the highest acceptable code

BC15:ldy #&FF**BC17:sty &3B**

Make X and Y point to the parameter area

BC19:iny**BC1A:ldx #&37**

Call OSWORD &00

BC1C:tya**BC1D:jsr &FFF1**

Zero COUNT and exit if 'Escape' was not pressed

BC20:bcc &BC28

Give the 'Escape' error message

BC22:jmp &9838**Move to a new line**

Call OSNEWL

BC25:jsr &FFE7

Zero COUNT

BC28:lda #&0
BC2A:sta &1E

Exit

BC2C:rts

Delete a line from the program

On entry, the line number of the line to be deleted should be in the IAC

Search for the line and exit with C = 1 if it does not exist

BC2D:jsr &9970
BC30:bcs &BC80

Subtract two from the address of the line, and save the result in (&37), (&3D) and (&12)
 (TOP)

BC32:lda &3D
BC34:sbc #&2
BC36:sta &37
BC38:sta &3D
BC3A:sta &12
BC3C:lda &3E
BC3E:sbc #&0
BC40:sta &38
BC42:sta &13
BC44:sta &3E

Get the length of the line

BC46:ldy #&3
BC48:lda (&37),Y

Add this to (&37)

BC4A:clc
BC4B:adc &37
BC4D:sta &37
BC4F:bcc &BC53
BC51:inc &38

Point to the first character of the following line

BC53:ldy #&0

Get the next character of the following line

BC55:lda (&37),Y

Move it down over the current line

BC57:sta (&12),Y

Stop the process when a carriage return is encountered

BC59:cmp #&D
BC5B:beq &BC66

Increment the offset

BC5D:iny
BC5E:bne &BC55

Increment the MSBs of the pointers if Y becomes zero

BC60:inc &38

BC62:inc &13

BC64:bne &BC55

Increment the pointers past the carriage return

BC66:iny

BC67:bne &BC6D

BC69:inc &38

BC6B:inc &13

Transfer the MSB of the next line's line number

BC6D:lda (&37),Y

BC6F:sta (&12),Y

Exit if the line number indicates the end of the program

BC71:bmi &BC7C

Increment the pointers and transfer the LSB of the line number

BC73:jsr &BC81

Increment the pointers and transfer the length of the line

BC76:jsr &BC81

Continue the process

BC79:jmp &BC5D

Update TOP

BC7C:jsr &BE92

Clear the carry flag to indicate success

BC7F:clc

Exit

BC80:rts

Increment the offset and the MSBs if required

BC81:iny

BC82:bne &BC88

BC84:inc &13

BC86:inc &38

Transfer a single byte

BC88:lda (&37),Y

BC8A:sta (&12),Y

Exit

BC8C:rts

Insert a line into the program

On entry the line number of the line should be in the IAC. Y must contain the offset into the keyboard buffer of the first textual character of the line (in other words, ignoring the line number)

Save the offset

BC8D:sty &3B

Delete the line if necessary

BC8F:jsr &BC2D

Store 7 into &3C. This makes (&3B) point to the first character of the line

BC92:ldy #&7

BC94:sty &3C

Exit if the first character of the line is a carriage return. This is how you can delete a line just by typing its line number

BC96:ldy #&0

BC98:lda #&D

BC9A:cmp (&3B),Y

BC9C:beq &BD10

Continue incrementing Y until a carriage return is encountered

BC9E:iny

BC9F:cmp (&3B),Y

BCA1:bne &BC9E

Add three to Y, to allow for the line number and the length of the line byte. This makes Y contain the length of the line

BCA3:iny

BCA4:iny

BCA5:iny

Save it in &3F

BCA6:sty &3F

Increment it again, without incrementing Y, to allow for the carriage return at the end of the line

BCA8:inc &3F

Make (&39) point to TOP

BCAA:lda &12

BCAC:sta &39

BCAE:lda &13

BCB0:sta &3A

Update TOP with Y

BCB2:jsr &BE92

Store the new value of TOP in (&37)

BCB5:sta &37

BCB7:lda &13

BCB9:sta &38

Decrement the line length

BCBB:dey

Check the new TOP against HIMEM

BCBC:lda &6

BCBE:cmp &12

BCC0:lda &7

BCC2:sbc &13

If there is enough room, skip the error message

BCC4:bcs &BCD6

Check for 'Bad program'

BCC6:jsr &BE6F

Carry out a CLEAR operation

BCC9:jsr &BD20

Give the 'LINE space' error message

BCCC:brk

BCCD:00-

BCCE:86-

BCCF:20-

BCD0:73-s

BCD1:70-p

BCD2:61-a

BCD3:63-c

BCD4:65-e

BCD5:00-

Move the top byte up to make room

BCD6:lda (&39),Y

BCD8:sta (&37),Y

Decrement the MSB of the pointers if Y is zero

BCDA:tya

BCDB:bne &BCE1

BCDD:dec &3A

BCDF:dec &38

Decrement the offset of the pointers

BCE1:dey

Subtract the lower pointer address from the start address of the line

BCE2:tya

BCE3:adc &39

BCE5:ldx &3A

BCE7:bcc &BCEA

BCE9:inx

BCEA:cmp &3D

BCEC:txa

BCED:sbc &3E

If enough bytes have not been moved, go back and continue moving them

BCEF:bcs &BCD6

Store the line number of the line

BCF1:sec

BCF2:ldy #&1

BCF4:lda &2B

BCF6:sta (&3D),Y

BCF8:iny

BCF9:lda &2A

BCFB:sta (&3D),Y

Store the length of the line

BCFD:iny

BCFE:lda &3F

BD00:sta (&3D),Y
 Add Y to (&3D)
BD02:jsr &BE56
 Initialise the loop counter
BD05:ldy #&FF
 Move the text of the line from the buffer to the program area
BD07:iny
BD08:lda (&3B),Y
BD0A:sta (&3D),Y
 Until a carriage return is encountered
BD0C:cmp #&D
BD0E:bne &BD07
 Exit
BD10:rts

RUN statement routine

Check for the end of the statement
BD11:jsr &9857
 Clear the variable catalogue and the various stacks
BD14:jsr &BD20
 Set PTR#1 to PAGE
BD17:lda &18
BD19:sta &C
BD1B:stx &B
 Execute from PTR#1 onwards
BD1D:jmp &8B0B

Clear variables

Set LOMEM and VARTOP to TOP
BD20:lda &12
BD22:sta &0
BD24:sta &2
BD26:lda &13
BD28:sta &1
BD2A:sta &3
 Clear the various BASIC stacks
BD2C:jsr &BD3A
 Indicate that 128 bytes of the variable catalogue need to be cleared
BD2F:ldx #&80
 Clear them by storing zero
BD31:lda #&0
 Store zero in the current catalogue location
BD33:sta &47F,X
 Continue the process for the rest of the locations

BD36:dex

BD37:bne &BD33

Exit

BD39:rts

Clear the BASIC stacks

This routine clears the main BASIC stack, sets the DATA pointer to PAGE, and clears the REPEAT, FOR and GOSUB stacks

Get PAGE into A

BD3A:lda &18

Save it as the MSB of the data pointer

BD3C:sta &1D

Set the stack pointer to HIMEM

BD3E:lda &6

BD40:sta &4

BD42:lda &7

BD44:sta &5

Zero the REPEAT, FOR and GOSUB stack pointers

BD46:lda #&0

BD48:sta &24

BD4A:sta &26

BD4C:sta &25

Zero the LSB of the DATA pointer

BD4E:sta &1C

Exit

BD50:rts

Push FAC # 1 on the BASIC stack

Lower the stack pointer by five bytes

BD51:lda &4

BD53:sec

BD54:sbc #&5

BD56:jsr &BE2E

Save the exponent

BD59:ldy #&0

BD5B:lda &30

BD5D:sta (&4),Y

BD5F:iny

Save the MSB of the mantissa combined with the sign bit

BD60:lda &2E

BD62:and #&80

BD64:sta &2E

BD66:lda &31

BD68:and #&7F

BD6A:ora &2E

BD6C:sta (&4),Y

BD6E:iny

Save the rest of the mantissa

BD6F:lda &32

BD71:sta (&4),Y

BD73:iny

BD74:lda &33

BD76:sta (&4),Y

BD78:iny

BD79:lda &34

BD7B:sta (&4),Y

Exit

BD7D:rts

Remove real number from the stack

After removal, (&4B) points to the number

Save the current LSB of the stack pointer in &4B

BD7E:lda &4

BD80:clc

BD81:sta &4B

Add five to it and replace it

BD83:adc #&5

BD85:sta &4

Set &4C to the MSB of the stack pointer

BD87:lda &5

BD89:sta &4C

Adjust it and replace it

BD8B:adc #&0

BD8D:sta &5

Exit

BD8F:rts

Push integer, real or string

This routine pushes either the number in FAC#1, the IAC or the string in the string buffer, according to the flag settings

Push a string if the type is string

BD90:beq &BDB2

Push a real number if the type is real

BD92:bmi &BD51

Fall into the integer push routine

Push integer

This routine pushes the integer in the IAC onto the BASIC stack

Subtract four from the stack pointer

BD94:lda &4

BD96:sec

BD97:sbc #&4

BD99:jsr &BE2E

Move the integer onto the stack

BD9C:ldy #&3

BD9E:lda &2D

BDA0:sta (&4),Y

BDA2:dey

BDA3:lda &2C

BDA5:sta (&4),Y

BDA7:dey

BDA8:lda &2B

BDAA:sta (&4),Y

BDAC:dey

BDAD:lda &2A

BDAF:sta (&4),Y

Exit

BDB1:rts

Push string

This routine pushes the string in the string buffer onto the BASIC stack

Subtract the length of the string to be pushed plus one from the stack pointer

BDB2:clc

BDB3:lda &4

BDB5:sbc &36

BDB7:jsr &BE2E

If the string is null, don't push it

BDBA:ldy &36

BDBC:beq &BDC6

Push the characters of the string

BDBE:lda &5FF,Y

BDC1:sta (&4),Y

BDC3:dey

BDC4:bne &BDBE

Save the length of the string as the bottom byte

BDC6:lda &36

BDC8:sta (&4),Y

Exit

BDCA:rts

Pull string

Get the length of the string

BDCB:ldy #&0

BDCD:lda (&4),Y

BDCF:sta &36

Don't pull the characters of the string if it is null

BDD1:beq &BDDC

Pull the characters off the stack into the buffer

BDD3:tay

BDD4:lda (&4),Y

BDD6:sta &5FF,Y

BDD9:dey

BDDA:bne &BDD4

Discard string

This routine removes the string on the top of the stack

Add the length of the string plus one to the stack pointer

BDDC:ldy #&0

BDDE:lda (&4),Y

BDE0:sec

BDE1:adc &4

BDE3:sta &4

BDE5:bcc &BE0A

BDE7:inc &5

Exit

BDE9:rts

Pull integer

Move the bytes of the integer from the stack to the IAC

BDEA:ldy #&3

BDEC:lda (&4),Y

BDEE:sta &2D

BDF0:dey

BDF1:lda (&4),Y

BDF3:sta &2C

BDF5:dey

BDF6:lda (&4),Y

BDF8:sta &2B

BDF A:dey

BDFB:lda (&4),Y

BDFD:sta &2A

Discard integer

This routine discards the integer on the top of the stack by adding four to the the stack pointer

Add four to the stack pointer

BDF F:clc

BE00:lda &4

```

BE02:adc  #&4
BE04:sta  &4
BE06:bcc  &BE0A
BE08:inc  &5

```

Exit

```
BE0A:rts
```

Pull integer to &37 onwards

Indicate address as &37

```
BE0B:ldx  #&37
```

Pull integer to &00XX onwards

This routine pulls the integer on the top of the stack to the four zero page locations indicated by X

Move the MSB

```

BE0D:ldy  #&3
BE0F:lda  (&4),Y
BE11:sta  &3,X

```

Move the next byte

```

BE13:dey
BE14:lda  (&4),Y
BE16:sta  &2,X

```

Move the next byte

```

BE18:dey
BE19:lda  (&4),Y
BE1B:sta  &1,X

```

Move the LSB

```

BE1D:dey
BE1E:lda  (&4),Y
BE20:sta  &0,X

```

Add four to the stack pointer

```

BE22:clc
BE23:lda  &4
BE25:adc  #&4
BE27:sta  &4
BE29:bcc  &BE0A
BE2B:inc  &5

```

Exit

```
BE2D:rts
```

Lower the stack, checking for 'No room'

On entry, A should contain the new value for the LSB of the stack pointer. The carry flag should be set/reset according to the subtraction which lowered the LSB

Save the LSB of the pointer

BE2E:sta &4

Decrement the MSB if needed

BE30:bcs &BE34

BE32:dec &5

Give a 'No room' error message if there is not enough room for the projected stack pointer

BE34:ldy &5

BE36:cpy &3

BE38:bcc &BE41

BE3A:bne &BE40

BE3C:cmp &2

BE3E:bcc &BE41

Exit

BE40:rts

Direct jump to the 'No room' error message

BE41:jmp &8CB7

Copy the IAC to &00XX

This routine copies the IAC to the four zero page locations indicated by X

Move the bytes

BE44:lda &2A

BE46:sta &0,X

BE48:lda &2B

BE4A:sta &1,X

BE4C:lda &2C

BE4E:sta &2,X

BE50:lda &2D

BE52:sta &3,X

Exit

BE54:rts

Add Y to &3D,&3E

Add Y to the locations

BE55:clc

BE56:tya

BE57:adc &3D

BE59:sta &3D

BE5B:bcc &BE5F

BE5D:inc &3E

Set Y to one

BE5F:ldy #&1

Exit

BE61:rts

Load a new program

Notice that this routine ends with an RTS

Set up the parameter block

BE62:jsr &BEDD

Make Y zero

BE65:tay

Indicate that a LOAD operation is required

BE66:lda #&FF

Tell OSFILE to load the file to the address in the parameter block

BE68:sty &3D

Make X point to the parameter block

BE6A:ldx #&37

Call OSFILE

BE6C:jsr &FFDD

Fall into a check for 'Bad program'

Check for a 'Bad program'

Set TOP to PAGE. TOP will then be used as a pointer to step through the program

BE6F:lda &18

BE71:sta &13

BE73:ldy #&0

BE75:sty &12

Increment the offset of TOP, to allow for the first DEY instruction

BE77:iny

Make Y point to the carriage return at the end of the line

BE78:dey

Give a 'Bad program' message if the carriage return is not present

BE79:lda (&12),Y

BE7B:cmp #&D

BE7D:bne &BE9E

Get the MSB of the line number

BE7F:iny

BE80:lda (&12),Y

End the search if the MSB signals the end of the program

BE82:bmi &BE90

Get the length of the line

BE84:ldy #&3

BE86:lda (&12),Y

Give a 'Bad program' message if the length of the line is zero

BE88:beq &BE9E

Add the length of the line to TOP, to make it point to the next line

BE8A:clc

BE8B:jsr &BE93

Continue scanning the next line

BE8E:bne &BE78

Add Y to LSB of TOP

BE90:iny
BE91:clc
BE92:tya
BE93:adc &12
BE95:sta &12

Increment the MSB if necessary

BE97:bcc &BE9B
BE99:inc &13

Make the offset be 1

BE9B:ldy #&1

Exit

BE9D:rts

Print the 'Bad program' message

BE9E:jsr &BFCF

(The disassembler cannot cope with calls to &BFCF, and so encodes the first three characters as a proper instruction. The characters present are a carriage return followed by the letters 'Ba')

BEA1:ora &6142
BEA4:64-d
BEA5:20-
BEA6:70-p
BEA7:72-r
BEA8:6F-o
BEA9:67-g
BEAA:72-r
BEAB:61-a
BEAC:6D-m
BEAD:0D-

The message is terminated by NOP instruction, because the JMP instruction does not have its top bit set

BEAE:EA-j

Jump to the cold start entry

BEAF:4C-L
BEB0:F6-v
BEB1:8A-

Place a CR at the end of a string

This routine adds a carriage return to the end of the string in the string buffer, and makes (&37) point to the string

Make (&37) point to the string

BEB2:lda #&0
BEB4:sta &37
BEB6:lda #&6

BEBA:ldy &36
BEBC:lda #&D
BEBE:sta &600,Y

Exit

BEC1:rts

OSCLI statement routine

Evaluate the string argument and place a carriage return at the end of it

BEC2:jsr &BED2

Point to the string buffer

BEC5:ldx #&0

BEC7:ldy #&6

Call OSCLI

BEC9:jsr &FFF7

Exit

BECC:jmp &8B9B

Direct jump to the 'Type mismatch' error message

BECE:jmp &8C0E

Evaluate string and add a CR

Evaluate the expression

BED2:jsr &9B1D

Give a 'Type mismatch' error if it is not a string

BED5:bne &BECF

Add a carriage return to the end of the string

BED7:jsr &BEB2

Check for the end of the statement

BEDA:jmp &984C

Set up filename and load/save address

This routine sets up the parameter area for a call to OSFILE

Deal with the filename

BEDD:jsr &BED2

Make the load address of the file be PAGE

BEE0:dey

BEE1:sty &39

BEE3:lda &18

BEE5:sta &3A

Use the machine higher order address as the top two bytes of the address

BEE7:lda #&82

BEE9:jsr &FFF4

BEEC:stx &3B

BEEE:sty &3C

Exit

BEF0:lda #&0

BEF2:rts

SAVE command routine

Check for a 'Bad program'

BEF3:jsr &BE6F

Set the end address as TOP

BEF6:lda &12**BEF8:sta &45****BEFA:lda &13****BEFC:sta &46**

Set the execution address as &8023

BEFE:lda #&23**BF00:sta &3D****BF02:lda #&80****BF04:sta &3E**

Use PAGE as the start address

BF06:lda &18**BF08:sta &42**

Set up the parameter area

BF0A:jsr &BEDD

Fill in the higher order addresses

BF0D:stx &3F**BF0F:sty &40****BF11:stx &43****BF13:sty &44****BF15:stx &47****BF17:sty &48**

Zero the LSB of the start address

BF19:sta &41

Make X and Y point to the parameter area

BF1B:tay**BF1C:ldx #&37**

Call OSFILE

BF1E:jsr &FFDD

Exit

BF21:jmp &8B9B**LOAD command routine**

Load the new program

BF24:jsr &BE62

Exit

BF27:jmp &8AF3**CHAIN statement routine**

Load the new program

BF2A:jsr &BE62

Join the code for the RUN command

BF2D:jmp &BD14

Join the code for the RUN command

BF2D: jmp &BD14

PTR statement routine

This routine deals with PTR when it is used on the left of an assignment statement

Get the file handle

BF30: jsr &BFA9

Save it on the stack

BF33: pha

Check the equals sign and evaluate the expression following it

BF34: jsr &9813

Ensure the result is an integer

BF37: jsr &92EE

Retrieve the file handle and put it in Y

BF3A: pla

BF3B: tay

Point to the IAC to contain the new PTR value

BF3C: ldx #&2A

Indicate the 'set pointer' command

BF3E: lda #&1

Call OSARGS

BF40: jsr &FFDA

Exit

BF43: jmp &8B9B

EXT function routine

Set the carry flag and fall into the PTR routine

BF46: sec

PTR function routine

This routine copes with PTR when it appears in an expression. When this routine is entered, the carry flag is always cleared, but when it is entered from EXT, the carry flag is set. This is how the same routine can serve two keywords

Place the carry flag into bit 6 of the accumulator

BF47: lda #&0

BF49: rol A

BF4A: rol A

Save this status on the machine stack

BF4B: pha

Get the file handle

BF4C: jsr &BFB5

Make X point to the IAC as the destination

BF4F: ldx #&2A

Retrieve the status and use it as the command for OSARGS

BF51: pla

Call OSARGS

BF52:jsr &FFDA

Indicate the result is an integer

BF55:lda #&40

Exit

BF57:rts

BPUT statement routine

Get the file handle

BF58:jsr &BFA9

Save it on the stack

BF5B:pha

Skip the comma following the handle

BF5C:jsr &8AAE

Evaluate the expression and check for the end of the line

BF5F:jsr &9849

Ensure the result is an integer

BF62:jsr &92EE

Retrieve the file handle and place it in Y

BF65:pla

BF66:tay

Get the byte to BPUT

BF67:lda &2A

Call OSBPUT

BF69:jsr &FFD4

Exit

BF6C:jmp &8B9B

BGET function routine

Get the file handle

BF6F:jsr &BFB5

Get a byte from the file

BF72:jsr &FFD7

Use the byte to fill up to the IAC and exit

BF75:jmp &AED8

OPENIN function routine

Indicate that a file is to be opened for input and join the code for OPENUP

BF78:lda #&40

BF7A:bne &BF82

OPENOUT function routine

Indicate that a file is to be opened for output and join the code for OPENUP

BF7C:lda #&80

BF7E:bne &BF82

OPENUP function routine

Indicate that a file is to be opened for update

BF80:lda #&C0

Save the type of opening

BF82:pha

Evaluate the filename

BF83:jsr &ADEC

Give a 'Type mismatch' error if the result is not a string

BF86:bne &BF96

Place a carriage return at the end of the string

BF88:jsr &BEBA

Point to the string buffer with X and Y

BF8B:ldx #&0

BF8D:ldy #&6

Retrieve the type of opening required

BF8F:pla

Open the file

BF90:jsr &FFCE

Fill the IAC with the file handle and exit

BF93:jmp &AED8

Direct jump to the 'Type mismatch' error

BF96:jmp &8C0E

CLOSE statement routine

Get the file handle

BF99:jsr &BFA9

Check for the end of the statement

BF9C:jsr &9852

Get the file handle into Y

BF9F:ldy &2A

Indicate a file closure is required

BFA1:lda #&0

Close the file(s)

BFA3:jsr &FFCE

Exit

BFA6:jmp &8B9B

Get a file handle

Set PTR#2 to PTR#1

BFA9:lda &A

BFAB:sta &1B

BFAD:lda &B

BF AF:sta &19

BFB1:lda &C

BFB3:sta &1A

Give a 'Missing #' error if the next character is not a hash

BFB5:jsr &8A8C

BFB8:cmp #&23

BFBA:bne &BFC3

Evaluate the handle as an integer

BFBC:jsr &92E3

Get the handle into Y

BFBF:ldy &2A

And A

BFC1:tya

Exit

BFC2:rts

BFC3:brk

BFC4:2D--

BFC5:4D-M

BFC6:69-i

BFC7:73-s

BFC8:73-s

BFC9:69-i

BFCA:6E-n

BFCB:67-g

BFCC:20-

BFCD:23-#

BFCE:00-

Print a string

This routine prints the ASCII characters following the JSR &BFCE. The string is terminated with a byte with bit 7 set. This byte will be executed as a 6502 instruction

Retrieve the address of the start of the message

BFCF:pla

BFD0:sta &37

BFD2:pla

BFD3:sta &38

Set the pointer

BFD5:ldy #&0

Jump into the routine

BFD7:beq &BFDC

Print the character

BFD9:jsr &FFE3

Get the next character

BFDC:jsr &894B

Return for a new character if the top bit is not set

BFDF:bpl &BFD9

Return to the calling position

BFE1: jmp (&37)

REPORT statement routine

Check for the end of the statement

BFE4: jsr &9857

Move to a new line

BFE7: jsr &BC25

Point to the first character of the error message

BFEA: ldy #&1

Get the current character of the error message

BFEC: lda (&FD), Y

Exit if it is the zero byte marking the end of the message

BFEE: beq &BFF6

Print the character

BFF0: jsr &B50E

Point to the next character

BFF3: iny

Go back to output it

BFF4: bne &BFEC

Exit

BFF6: jmp &8B9B

Signing off

These seven bytes were inserted to remind people disassembling the BASIC ROM that it was written by Roger Wilson.

BFF9: 00-

BFFA: 52-R

BFFB: 6F-o

BFFC: 67-g

BFFD: 65-e

BFFE: 72-r

BFFF: 00-

Index

6

- 6502 addressing modes 11
- 6502 registers 10

A

- ADC instruction 14
- AND instruction 15
- AND operator 43
- ASL instruction 15
- Accumulator addressing 11
- Addressing modes 11
- Alphabetical index of BASIC ROM . . 178

B

- BASIC addition routine 59
- BASIC multiplication routine 63
- BASIC normalisation routine 57
- BASIC square root routine 65
- BCC instruction 16
- BCS instruction 17
- BEQ instruction 17
- BIT instruction 17
- BMI instruction 18
- BNE instruction 18
- BPL instruction 18
- BRK instruction 18
- BVC instruction 18
- BVS instruction 18
- Boolean arithmetic 43

C

- CLC instruction 18
- CLD instruction 18
- CLI instruction 18

- CLV instruction 19
- CMP instruction 19
- CPX instruction 19
- CPY instruction 19

D

- DEC instruction 20
- DEX instruction 20
- DEY instruction 20
- Direct addressing 11
- Disassembler 161
- Diversion: Blowing up the screen . . . 68
- Diversion: Bresenham meets Moire . . 66
- Diversion: DNA 158
- Diversion: Double vertical resolution . 83
- Diversion: Drawing a mexican hat . . . 27
- Diversion: Drawing a milk splash . . . 41
- Diversion: Drawing on the tube 81
- Diversion: Fan 159
- Diversion: Faster cursor keys 49
- Diversion: Overlaid sine waves 29
- Diversion: Screen oddity 114

E

- EOR instruction 20
- EOR operator 44
- Exponentiation 40

F

- FROTH vocabulary 113
- Floating point accumulators 56
- Floating point addition 54
- Floating point division 56

Floating point multiplication.....	56
Floating point representation.....	54

G

GCOL.....	45
-----------	----

I

INC instruction.....	20
INX instruction.....	20
INY instruction.....	20
Immediate addressing.....	11
Implied addressing.....	11
Index (alphabetical) of BASIC ROM.....	178
Indexed addressing.....	12
Indirect addressing.....	12
Inherent addressing.....	11

J

JMP instruction.....	20
JSR instruction.....	20

K

Keyword table.....	174
--------------------	-----

L

LDA instruction.....	21
LDX instruction.....	21
LDY instruction.....	21
LSR instruction.....	22
Line number storage.....	174

M

Memory usage.....	170
Multiplication algorithm.....	34

N

NOP instruction.....	22
NOT operator.....	44
Normalisation.....	53

O

OR operator.....	43
ORA instruction.....	22

P

PHA instruction.....	22
PHP instruction.....	22
PLA instruction.....	22
PLP instruction.....	23
PROC/FN handling.....	191
Page zero usage.....	170
Post-indexed indirect addressing.....	12
Pre-indexed indirect addressing.....	12
Priority in expressions.....	70
Priority in graphics.....	46

R

ROL instruction.....	23
ROR instruction.....	23
RTI instruction.....	23
RTS instruction.....	24
Registers.....	10
Relative addressing.....	13
Reverse Polish Notation.....	70

S

SBC instruction.....	24
SEC instruction.....	24
SED instruction.....	24
SEI instruction.....	25
SLUG syntax.....	118
STA instruction.....	25
STX instruction.....	25
STY instruction.....	25
Scientific notation.....	52
Status register.....	13

T

TAX instruction.....	25
TAY instruction.....	25
TSX instruction.....	25
TXA instruction.....	26

TXS instruction.....	26	Z
TYA instruction.....	26	
Two's complement arithmetic.....	32	Zero page usage.....170

V

Variable storage.....	173
-----------------------	-----

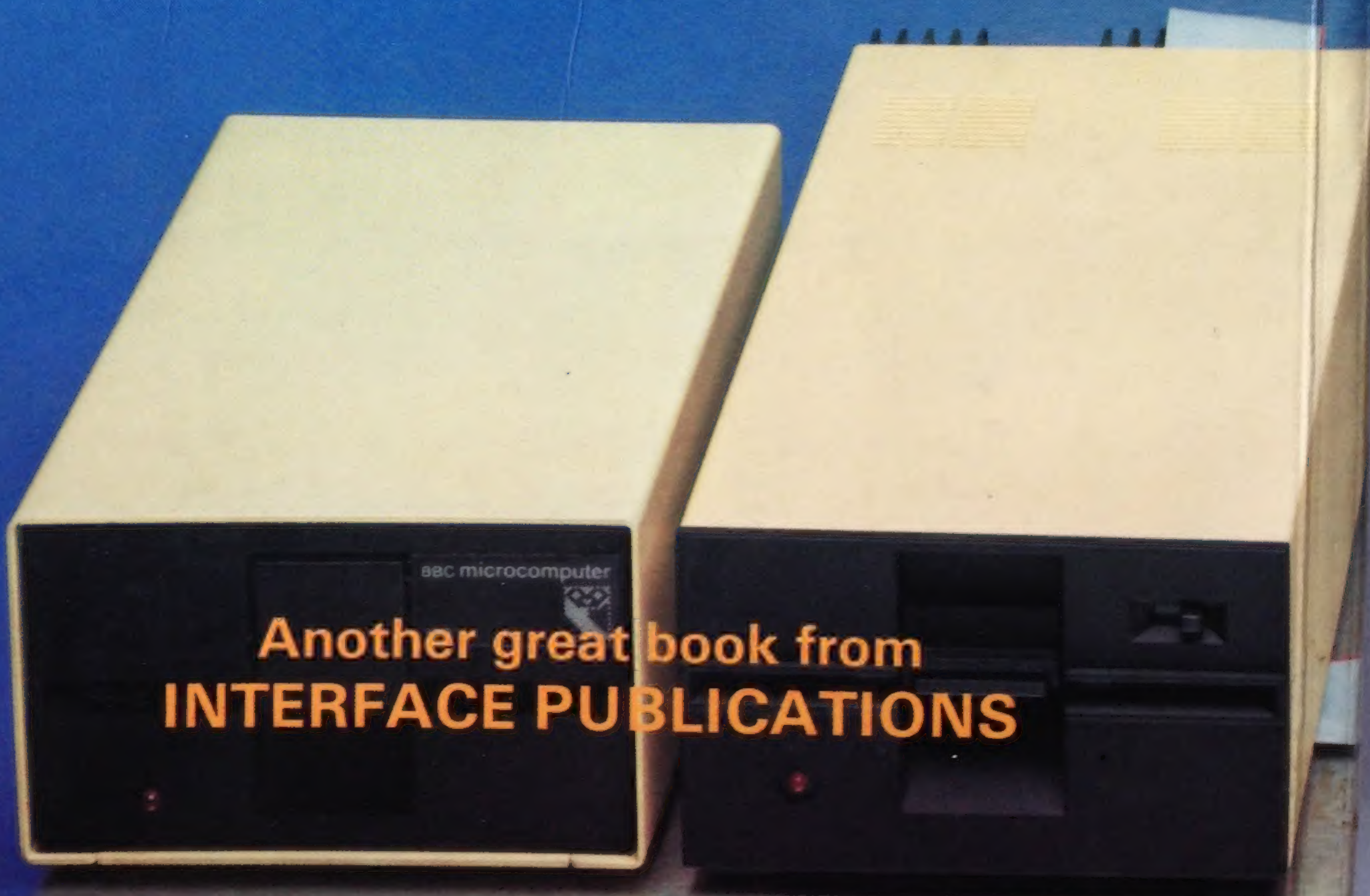
THE BBC MICRO COMPENDIUM By Jeremy Ruston

This book is a complete tutorial on advanced programming for the BBC Micro. Major topics covered include:

- Assembly language programming
- Computer arithmetic, including full details of floating point algorithms
- Recursive programming
- How to increase the vertical screen resolution to 512 pixels with no hardware modifications
- An intelligent disassembler that can work out the areas of a program that are data

The two most substantial sections cover a complete and carefully annotated disassembly of the BASIC ROM at the heart of the BBC Micro and the theory and practice of languages in general. The latter section includes two complete compilers. One compiles FROTH, a threaded language resembling FORTH and the other compiles a new language called SLUG, which is a block structured language based on Algol, BCPL and Pascal. Both these languages can run up to 100 times faster than BBC BASIC.

The BBC Micro Compendium is written by Jeremy Ruston, bestselling author of 'The BBC Micro Revealed', 'Learn Pascal on your BASIC Micro' and co-author of 'The Book of Listings'.



**Another great book from
INTERFACE PUBLICATIONS**